FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Um programa de elementos finitos em ambiente aberto

Tiago Rodrigues Carriço Mendes

Programa de Mestrado de Engenharia Civil

Orientador: Professor João Macedo

Outubro de 2018

Versão Final

Um programa de elementos finitos em ambiente aberto

Tiago Rodrigues Carriço Mendes

Programa de Mestrado de Engenharia Civil

Outubro de 2018
Versão Final

Resumo

O avanço da tecnologia e o seu envolvimento com a sociedade tem sido um fator com cada vez mais realçe na área das ciências e da engenharia. A necessidade que o ser humano demonstra em recorrer às mais recentes e potentes ferramentas de cálculo tem vindo a crescer e não demonstra sinais de paragem. Assim sendo, nos últimos anos, no que toca à engenharia civil, foi possível observar um grande surgimento de programas de análise de estruturas que facilitam o cálculo das mesmas e que permitem poupar muito tempo e trabalho.

Ao longo desta dissertação procurou-se abordar o tema de como a tecnologia informática se consegue inserir no mundo da engenharia civil, mais concretamente no cálculo de estruturas através do método dos elementos finitos. Neste caso, esta inserção diz respeito à criação de *software* de elementos finitos e a sua consequente disponibilização numa plataforma *open source* para que o desenvolvimento do mesmo seja possível. Como esta fase é apenas a fase inicial da criação de um programa, é expectável que surjam contribuições de várias pessoas ao longo dos tempos e que um dia, o programa esteja completo.

Para que o programa esteja apto a receber contribuições, este necessita de uma dada estruturação como é explicado ao longo do trabalho, pois a entrada destas tem de ser natural. Para contribuir para o mesmo será necessário seguir uma série de passos também devidamente descritos para que, posteriormente, a contribuição em causa seja validada e possa ser incluída no programa. Por exemplo, no trabalho mostra-se como será simples a entrada de um novo tipo de elemento, sendo necessário apenas a criação de uma série de ficheiros com a especificidade desse mesmo elemento, que podem ser incluídos em apenas algumas horas, graças à estruturação previamente estabelecida. Para a validação destas contribuições são efetuados testes de aceitação de entrada que avaliam a qualidade do output do novo elemento submetido e comparam-na com valores reais.

PALAVRAS-CHAVE: *software*, *open source*, elementos finitos, *python*, estruturas

Abstract

The progress of technology and its involvement with society has been a factor with a growing highlight in the fields of science and engineering. The necessity that the human being shows to use the latest and powerful calculus tools has been growing and isn't showing any signs of stopping. That being said, in the last years, when it comes to civil engineering, it's been possible to observe a big appearance of structural analysis programs that aid the solving of these same structures and allow the saving of time and work.

Along this dissertation, it was made an approach on how the technology could insert itself in the civil engineering world, more specifically in the computing of structures using the finite element method. In this case, the basis of an open source finite element program was created. This is really the very initial stage of a open source program that hopefully will expand to a full and complete finite element program once the contributions from any other students from FEUP or from any other school start to enter in the program.

It was established a program structure that should be very easy to use and understand, in order to make natural the arrival of any contributions. To a successful contribution, the user must follow a specific series of steps so that, this same contribution is validated and included in the program. The validation of these entries is performed with the use of acceptance tests that evaluates, for instance, the arrival of a new type of element and compares its output with real values, obtained with a finite element software.

KEY-WORDS: software, open source, finite elements, structures

Agradecimentos

A toda a minha família pelo apoio incondicional, não só ao longo desta dissertação, mas também ao longo destes 5 anos.

À minha namorada, pela paciência e ajuda que me ofereceu em todo este processo.

A todos os meus amigos que de certa forma puderam contribuir para todo o meu conhecimento e tudo que sou hoje.

Ao meu orientador, o professor João Macedo, que sempre disponibilizou o seu tempo para prestar qualquer ajuda e apoio neste trabalho. Todo o seu conhecimento e colaboração foram preponderantes para a conclusão desta dissertação.

Tiago Rodrigues Carriço Mendes

"You should be glad that bridge fell down.
I was planning to build thirteen more to that same design"

Isambard Kingdom Brunel

Conteúdo

Lista de Figuras

Abreviaturas e Símbolos

OSD	Open Source Definition
OSS	Open Source Software
OSI	Open Source Initiative
JSON	JavaScript Object Notation
XML	Extensible Markup Language
GUI	Graphical User Interface
CLI	Command Line Interface
CSV	Comma-Separated Values
DXF	Drawing Interchange Format
MATLAB	Matrix Laboratory
FEUP	Faculdade de Engenharia da Universidade do Porto

Capítulo 1

Introdução

1.1 Enquadramento Geral

Nos dias de hoje é facilmente percetível que o mundo está em constante mutação e que esta é cada vez mais rápida e acentuada. Esta mudança deve-se ao grande desenvolvimento tecnológico que se tem assistido no decorrer das últimas décadas. A acompanhar este crescimento, tem surgido também um crescente investimento na inovação e tecnologia por parte de empresas de todo o mundo, das mais diversas áreas, com o objetivo de conseguir responder às necessidades presentes no mercado. Como é fácil de prever, a engenharia, mais particularmente a engenharia civil, não foi exceção. Surgiu a necessidade da realização de enumeras tarefas de modo conciso e prático e, todo este desenvolvimento e aperfeiçoamento tecnológico emergente, que contém elementos capazes de perfazer uma análise e resolução dos mais variados problemas de uma maneira simples, será capaz de automatizar estas mesmas tarefas, na forma de um programa por exemplo, que terá sido produzido com este mesmo intuito.

Tendo então como objetivo o uso de ferramentas tecnológicas cada vez mais aptas à resolução de problemas de engenharia, que por norma representam uma sequência de múltiplos cálculos de elevada dificuldade e complexidade, da maneira mais económica possível, o mercado da computação e *software*, mais especificamente de *software* de análise estrutural, não tardou a crescer, tornando então cada vez mais possível e acessível a toda a gente a resolução de problemas de engenharia civil com muita mais precisão, rapidez e versatilidade.

Em meados do século XX começaram a surgir os primeiros traços daquilo que seriam os atuais programas de análise estrutural que tomam como base o método dos elementos finitos. O primeiro *software* comercial a utilizar o método dos elementos finitos foi desenvolvido pela NASA em parceria com a empresa MacNeal-Schwendler, após uma revisão anual do programa de pesquisa de dinâmica estrutural em 1964, chamado MSC Nastran. Desde então foram surgindo cada vez mais programas desta dimensão e com e versatilidade para se conseguirem enquadrar nas diversas áreas onde este método é aplicado, seja na mecânica, dinâmica dos fluidos, eletromagnetismo ou

no cálculo estrutural. Um grande exemplo deste tipo de programas e com grande popularidade, mesmo no ramo académico, é o ANSYS, usado também nas aulas da Unidade Curricular de Análise Avançada de Estruturas.

Incluídos nesta incomensurável lista de programas estão também os chamados de *software* livre, ou *Open Source Software*, muito utilizados por estudantes e até grandes empresas que procuram soluções específicas, uma vez que a edição e alteração do código é permitida.

1.2 Objetivos da Dissertação

Com um crescimento tecnológico tão acentuado e com uma necessidade acrescida de o acompanhar, o uso das melhores ferramentas que existem no mercado é imprescindível para que a área da engenharia de estruturas possa evoluir.

Tomando por base esta premissa, o presente trabalho visa a elaboração de um programa de elementos finitos em ambiente aberto, que como o próprio nome diz, estará disponível para qualquer pessoa. Usando as mais modernas e potentes ferramentas do ramo informático será possível a criação de uma plataforma, em nome da Faculdade de Engenharia da Universidade do Porto e do Departamento de Engenharia Civil, que tem como objetivo poder ser também desenvolvida por futuros estudantes do ramo de especialização de estruturas e por qualquer pessoa de qualquer parte do mundo. Este programa tem de ser estruturado de tal maneira que, qualquer contribuição com a informação respetiva a um novo elemento a ser adicionado tem de conseguir ser encaixada de forma natural.

Com a existência de uma plataforma deste nível, tanto a aprendizagem como o ensino deste método a futuros alunos poderá ser muito mais dinamizada. O conhecimento e domínio de uma linguagem de programação é também um fator cada vez mais valorizado no mercado de trabalho.

1.3 Estrutura da Dissertação

A presente dissertação está organizada por vários capítulos e subcapítulos que servem como guia e fornecem diretrizes específicas para como proceder à elaboração de um programa deste tipo e como poder contribuir para o mesmo.

Neste primeiro capítulo foi feito um enquadramento geral do tema em questão e de seguida apresentados os objetivos deste trabalho.

No capítulo 2 é abordada a maneira de como se deve criar um projeto *Open Source* e todas as ações e cuidados que se deve ter. É descrito tudo que o processo implica, quais os diferentes tipos de licenças existentes e qual a mais adequada ao projeto. É explicada também a maneira como

lidar com diferentes contribuições.

No capítulo 3 é demonstrada a maneira como o programa está estruturado de forma a aguentar com diferentes contribuições. Apresenta-se detalhadamente o que compõe cada fase do programa e de que maneira é que novas entradas podem ser encaixadas.

No capítulo 4 são explicadas as regras de submissão de um elemento. Demonstra-se que ficheiros são necessários à contribuição por parte do utilizador e, posteriormente, de que forma é que o elemento em questão pode ou não ser validado.

No capítulo 5 é apresentado um exemplo de uma possível contribuição no modelo e a maneira como este se insere no mesmo.

O capítulo 6 está destinado a todas as conclusões e considerações finais desta dissertação bem como todos os desenvolvimentos futuros que possam vir a surgir em prol deste projeto.

Capítulo 2

Open Source Software

Um *Open Source Software* (OSS) não é, nada mais nada menos, do que um programa que disponibiliza o seu source code a qualquer pessoa para que esta o possa ver, estudar, usar, modificar e distribuir com qualquer propósito ou objetivo. Hoje em dia, qualquer pessoa é capaz de recorrer a um programa open source para proveito próprio, mas nem sempre foi assim. Este conceito só começou a ser formalmente usado com a criação da chamada *Open Source Initiative*(OSI) nos finais da década de 90. Até aí, todo o programa que disponibilizava o seu source code era chamado de free software, mas este conceito foi mudado devido à ambiguidade do seu nome.

A comunidade em geral começou desde então a perceber que poderia ganhar muito ao recorrer a este tipo de software e são inúmeras as suas vantagens, tanto para o criador do próprio programa como para um simples contribuidor de um dado programa também. Começando pelo preço, é facilmente inteligível que estes tipos de programas são mais baratos do que um produto do mercado. Estima-se que por ano, o uso de um OSS ajude a poupar biliões de euros às pessoas envolvidas neste tipo de projeto. E nem é de espantar visto que este tipo de programas foi feito para ser acedido por todos, especialmente para aqueles que não têm dinheiro para comprar produtos comerciais. Geralmente, estes são grátis e não é necessário pagar por qualquer cópia adicional.

Fora as questões monetárias, prevalecem também vantagens relativas à solidez e durabilidade do programa. Existem atualmente projetos open source com milhares de contribuidores. É evidente que, subsistindo num trabalho deste tipo e dimensão, ocorrerá uma grande contribuição para o conhecimento de outros, mas também para enriquecimento do próprio programa. Genericamente falando, existe um apelo tácito para que as pessoas comentem e critiquem o trabalho para o qual estão a contribuir, para que ambas as partes possam ganhar com a situação. Estas ações culminarão com certeza num trabalho muito mais seguro e livre de erros. Se eventualmente algum erro for detetado, este poderá até ser partilhado com outros para que não se volte a repetir no futuro. Esta abordagem traz também uma grande estabilidade a um trabalho pois, mesmo que os seus criadores originais deixem de estar envolvidos, o trabalho pode continuar a proliferar. É sem dúvida um caminho que pode ser, para quem usufrui destes programas, muito flexível. Não havendo qualquer tipo de vínculo, os criadores e contribuidores são livres de tomarem o seu próprio rumo e decidirem o querem fazer com o produto.

Como se pôde verificar, são inúmeras as vantagens em seguir este caminho, no entanto, é necessário tomar algumas precauções. É imperativo ter em atenção que ao começar este tipo de projeto, este estará a ser aberto para toda a gente. Utilizadores maliciosos podem visualizar o código disponibilizado e procurar vulnerabilidades e falhas no sistema computacional com o intuito de o prejudicar com a instalação de malware, por exemplo. Se não houver um esforço e dedicação de analisar o código frequentemente com o intuito de eliminar qualquer tipo de malícia, não é possível assumir que o projeto é seguro.

Por fim, tendo em conta todos os aspetos positivos e negativos, é razoável afirmar que os benefícios das vantagens inerentes ao uso de OSS são bastante superiores ao risco implícito das desvantagens do mesmo. É uma ótima abordagem a seguir, porém, há que ser feita com cautela pois são muitos os fatores a ter em conta e se estes não forem bem abordados tudo será uma grande perda de tempo, energia e recursos. [1] [13]

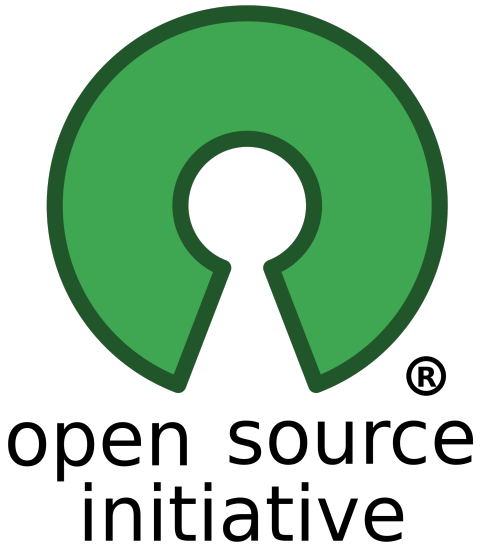

Figura 2.1: *Open Source Initiative*

2.1 *Open Source Definition*

Até ao momento, foi facilmente percetível que um OSS é capaz de fornecer e facilitar diversas coisas, no entanto, ainda não se falou em concreto no que é preciso para se começar um projeto deste tipo. É aqui que entra a *Open Source Definition* (OSD). Uma das primeiras tarefas da OSI foi de escrever a OSD e usá-la para criar uma lista de licenças aprovadas pela mesma. Open source não significa simplesmente um acesso direto ao source code, um OSS tem de cumprir uma série

de pontos, chamados de OSD, descritos de seguida [2].

- **Distribuição Livre.** A licença não deverá restringir ninguém de vender ou distribuir o software gratuitamente como componente de outro software ou não.
- **Source code.** O programa tem de incluir source code e tem de permitir a sua distribuição assim como na sua forma compilada. Se de alguma maneira, o programa não for distribuído com o seu source code, terá de existir obrigatoriamente um meio, devidamente publicitado, para o obter e este não deverá ter mais do que um simples custo de reprodução, preferencialmente um download via internet sem qualquer custo. O código tem de ser totalmente compreensível por qualquer programador. Código deliberadamente ofuscado não é permitido.
- **Trabalhos derivados.** A licença tem de permitir modificações e trabalhos variados e que estes sejam distribuídos sobre os mesmos termos da licença do software original.
- **Integridade do source code do autor.** A licença poderá restringir o source code de ser distribuído numa forma modificada apenas se a licença permitir a distribuição de patch files com o source code com o propósito de modificar o programa no momento da sua construção. A licença poderá também solicitar que trabalhos derivados tenham um nome ou número de versão diferentes do software original.
- **Não discriminação contra pessoas ou grupos.** A licença não pode discriminar qualquer pessoa ou grupo de pessoas.
- **Não discriminação contra áreas de atuação.** A licença não pode restringir ninguém de usar o programa numa área de atuação específica. Por exemplo, o programa não pode ser restringido de ser usado numa área comercial ou investigação médica.
- **Distribuição da licença.** Os direitos vinculados ao programa têm de ser aplicados a toda a gente cujo o programa foi redistribuído sem que seja preciso uma execução de uma licença adicional.
- **A licença não pode ser específica a um produto.** Os direitos vinculados ao programa não podem depender do facto do programa fazer parte de uma distribuição de software específica. Se o programa for extraído dessa distribuição e for usado ou distribuído dentro dos termos da licença do mesmo, toda a gente cujo o programa tenha sido distribuído deverá ter os mesmos direitos que aqueles que são garantidos em conjunção com a distribuição de software original.
- **A licença não pode restringir outro software.** A licença não pode impor restrições a outro software que seja distribuído juntamente com o software licenciado. Por exemplo, a licença não pode obrigar a que todos os outros programas distribuídos no mesmo meio de armazenamento sejam programas open source.

- **A licença tem de ser tecnologicamente neutra.** Nenhuma cláusula da licença pode estipular uma tecnologia individual ou estilo de interface.

2.2 Começar um projeto *Open Source*

É importante, para a criação de um projeto Open Source, que todos os pontos referidos anteriormente, respeitantes aos termos de distribuição, sejam cumpridos, porém, não é suficiente. Independentemente da altura em que é decidido que se vai começar um projeto deste tipo, todos estes devem conter a seguinte documentação:

- Licença *open source*
- *README*
- Diretrizes para contribuição
- Código de conduta

Um conhecimento vasto sobre estes documentos não é claramente obrigatório, mas é importante frisar os aspetos mais relevantes que podem ter uma influência considerável no projeto. Para que todo este processo seja uma boa experiência e que o projeto em si seja bem sucedido, todas estas componentes têm de ser tidas em conta pois irão ajudar significativamente na comunicação de todas as expectativas e objetivos estabelecidos pelos criadores, a organizar todas as contribuições e sobretudo a proteger os direitos legais de toda a gente envolvida [3].

2.2.0.1 Licença *Open Source*

Por definição, uma licença permite que um produto de software seja partilhado de diferentes maneiras para diferentes fins, quer sejam eles para investigação ou desenvolvimento e tudo isto sem qualquer tipo de repercussões. Regra geral, o licenciamento open source permite que o source code seja totalmente aberto e transparente e que este seja visto, utilizado e modificado por qualquer pessoa.

Para que seja possível escolher uma licença, há que garantir que a mesma é aprovada pela OSI, ou seja, a licença tem de estar de acordo com todos os pontos referentes à OSD e só assim esta será oficialmente considerada uma licença open source. Atualmente, a OSI tem a seu dispor mais de 80 licenças e todas estas acabam por se inserir em duas categorias: licenças copyleft e licenças permissivas. Tanto as permissivas como as copyleft permitem que os utilizadores possam livremente copiar, modificar e distribuir o software a estas associado. Portanto, a grande diferença entre as duas está nas condições sobre as quais os utilizadores podem usufruir do software.

O chamado copyleft é um método geral de fazer com que um programa ou qualquer tipo de trabalho seja livre e, além disso, exigir também que todas as extensões e versões modificadas desse mesmo programa sejam obrigatoriamente devolvidas. Simboliza uma licença que impede que o source code seja tomado como proprietário.

Consequentemente, no pólo oposto das licenças copyleft encontram-se as licenças permissivas que, na maioria dos casos, o uso e modificação do código é totalmente livre e não existe qualquer tipo de obrigação em apresentar as alterações que nele foram realizadas. O utilizador pode, dependendo dos termos da licença, apenas disponibilizar cópias do software em binário, com distribuição limitada e sem a necessidade de providenciar o source code modificado [4][5].

Para que este assunto seja facilmente compreendido e que haja outro tipo de sensibilidade para com o mesmo, pode-se recorrer às 4 liberdades. Segundo a Free Software Foundation, um programa para ser de software livre tem de permitir que os utilizadores do mesmo possuam 4 liberdades, definidas por esta mesma fundação. A classificação de uma licença como permissiva ou copyleft vai depender do número de liberdades que um dado utilizador possui. As liberdades são classificadas da seguinte maneira:

- **Liberdade 0** - Liberdade de correr o programa livremente, com qualquer propósito.
- **Liberdade 1** - Liberdade de estudar como é que o programa funciona e adaptá-lo às necessidades requeridas. O acesso ao source code é um requisito para isto.
- **Liberdade 2** - Liberdade de redistribuir cópias para que seja possível ajudar outros.
- **Liberdade 3** - Liberdade de distribuir cópias de versões modificadas. Impondo isto, estará a ser fornecida à comunidade a possibilidade de beneficiar destas mesmas alterações.

Uma licença copyleft certifica-se que qualquer pessoa que receba uma versão binária do software está no direito de receber a versão original, incluindo qualquer modificação, e essa pessoa, em troca, é obrigada a passar as 4 liberdades a qualquer outra pessoa cujo o código ou binários são passados.

No caso de ser uma licença permissiva, esta pode permitir que uma pessoa receba as 4 liberdades quando está a receber a cópia de um dado código mas depois, esta mesma pessoa, quando partilhar o software, tem todo o direito de escolher se quer passar as 4 liberdades ou qualquer uma individualmente[6].

Concluindo, e após uma análise de todos os fatores relacionados com a escolha de uma licença que podem influenciar o decorrer do projeto, a licença a usar neste trabalho em causa é a licença do MIT que corresponde a uma licença permissiva e das mais usadas em todo o mundo.

2.2.0.2 README

Um ficheiro README é um ficheiro que contém toda a informação útil sobre o software do programa. Regra geral, este aparece durante a instalação do programa ou é diretamente instalado. Tipicamente, este tipo de ficheiros contém uma série de instruções que ajudam e explicam como instalar o programa, como trabalhar com as funções mais básicas e o que este faz. Ocasionalmente, um ficheiro README pode conter também uma lista das mais recentes atualizações do programa e cuidados que se deva ter ou qualquer outro tipo de informação pertinente em relação ao uso do programa.

A leitura deste tipo de ficheiros é sempre vantajosa, especialmente quando se trata de projetos deste género pois, obviamente, vai existir sempre uma panóplia de questões significativas, ligadas ao funcionamento e uso do programa, que podem ser respondidas. É fundamental que seja compreendido por parte dos utilizadores o que é que o programa faz e porque é que é útil. Se quem está a contribuir não está minimamente ciente da envolvência e utilidade que pode ter, é sinal que não existe uma boa preparação por parte de quem criou o projeto para receber contribuidores. Conseguindo ter estas questões devidamente respondidas e explicitamente esclarecidas, pode-se partir para outros pontos, relativos a como se pode começar a estar envolvido no projeto e se por acaso for precisa ajuda, a quem é que se deve recorrer. É de salientar que muitas das questões abordadas neste ficheiro podem ser também debatidas num ficheiro de contribuição.

Se for pretendido, é possível discutir-se outro tipo interrogações mais pontuais sobre como serão lidadas as contribuições ao projeto, quais os objetivos do mesmo e até a existência de informação sobre a licença que se usou. Se por acaso não se estiver a aceitar contribuições ou o projeto simplesmente ainda não estiver pronto para tal, é importante que esta informação seja também descrita. A existência de um ficheiro README não é obrigatória, no entanto, é nítido que o uso deste é muito vantajoso [7].

2.2.0.3 Diretrizes para contribuição

Se existirem diretrizes específicas para contribuição, a participação de outros utilizadores no projeto torna-se consideravelmente mais fácil. Um ficheiro de contribuição tanto pode fornecer informação técnica específica como pode incluir todas as ideias do criador relativamente às suas expectativas de contribuição, ou seja, pode conter tópicos do género:

- Como reportar um bug ou qualquer tipo de falha.
- Como sugerir uma nova *feature*.
- Como configurar o ambiente e correr testes.

Com o passar do tempo poderão ser apresentadas questões menos técnicas e mais relacionadas com o criador e os seus objetivos mais pessoais. É do interesse do mesmo que seja percetível para toda a gente que tipo de contribuições é que são esperadas. Dependendo muito do projeto e da fase do mesmo, as contribuições necessárias vão variar. Outro ponto importante a ter em conta é ter objetivos bem definidos e um rumo totalmente traçado para que, consequentemente, quem estiver envolvido consiga trabalhar e produzir nesse sentido. Se por acaso houver dúvidas por parte de alguém externo que queira contribuir, será sempre pertinente a existência de informação que os guie a um contacto direto com o criador.

Ter uma documentação bem estruturada e organizada é um grande passo para que surjam contribuições de todo lado. São estas informações que podem fazer diferença no projeto pois o futuro e sucesso do mesmo depende diretamente das contribuições a ele associadas. Ao longo do tempo, com a intervenção dos variados utilizadores, surgirá consequentemente uma lista compilada de *frequently asked questions* que será certamente útil na utilização do programa[3].

2.2.0.4 Código de Conduta

Tal como o nome indica, o código de conduta não será mais do que um conjunto de regras que estabelecem o comportamento que deverá existir da parte de quem está envolvida no projeto. Se este código existir, garantimos que as pessoas envolvidas estarão protegidas, de certa forma, de qualquer tipo de atitude não produtiva perante o trabalho expectável de ser realizado. Se este tipo de ocorrências for evitado, o projeto será seguramente mais proativo e saudável. Deste modo, é impreterível que no código esteja explícito:

- Onde é que este tem efeito.
- A quem é que este pode ser aplicado.
- O que é que acontece se alguém quebrar este código de conduta.
- Como é que é possível reportar este tipo de infrações.

Mesmo não sendo do interesse dos criadores estar a escrever um código de conduta de raiz, não surgirá qualquer tipo de problema derivado do mesmo. O *Contributor Covenant*, por exemplo, é um código de conduta já estabelecido e usado por milhares de projetos open source. Tal como no caso das licenças open source, este foi feito para ser usado por toda a gente que assim o pretenda.

Mais importante do que a existência deste código é a demonstração de como este será aplicado. Se assim não acontecer, a imagem que estará a ser transmitida é que toda a informação e valores no código de conduta não são importantes ou respeitados pela comunidade. Portanto, adotando este tipo de postura, o projeto tornar-se-á consideravelmente mais sério e seguro[3].

Capítulo 3

Estrutura do projeto

Quando se está a começar, ou a pensar em começar um projeto de um programa, independentemente do seu conteúdo, a primeira coisa que deve ser tida em mente é a sua estrutura. Obviamente, o código escrito é o que fará o programa funcionar efetivamente, mas se o objetivo é que este corra de uma forma eficiente, então a sua estruturação não pode ser ignorada. Olhando para o conteúdo de um programa genérico, este será composto por dezenas de ficheiros e o objetivo principal é que a sua organização seja feita de tal maneira que estes consigam interagir entre si de uma maneira perfeita e limpa, só pela maneira como estão estruturados.

Um programa bem estruturado e organizado é meio passo para um projeto mais sustentável. Toda esta disposição dos ficheiros será um benefício não só para quem está a desenvolver atualmente o projeto, mas também para quem futuramente queira contribuir e trabalhar, pois a sua compreensão e leitura será bem mais fácil. Neste caso, num projeto *Open Source* de elementos finitos, o programa tem de estar organizado de tal maneira que as contribuições sejam incluídas de uma forma natural. A transversalidade é a chave do sucesso para um projeto desta dimensão.

Para que se possa então organizar o projeto em questão é preciso ter conhecimento de como o fazer e o que é que deve ser incluído no mesmo. Felizmente, para uma pessoa que esteja a começar um projeto pela primeira vez, existem várias diretrizes e passos a seguir que garantem o bom funcionamento do programa. Esta abordagem acaba por ser universal porque quase todos os projetos existentes, fora um detalhe ou outro que seja específico do mesmo, acabam por ser organizados da mesma maneira.

Na sua estrutura mais básica, um projeto é composto por 3 grandes partes que constituem o corpo do programa. A primeira, que diz respeito ao *Input*, é onde o programa recebe toda a informação necessária para executar as tarefas designadas, neste caso, o cálculo de um elemento específico. A segunda, é referente ao cálculo do elemento. Todos os cálculos respeitantes à geometria, material e rigidez do elemento são efetuados nesta parte. Por fim, com todos os cálculos realizados e toda a informação disponível, a terceira parte é definida pelo *Output* que não é, nada mais nada menos, do que a apresentação dos resultados obtidos.

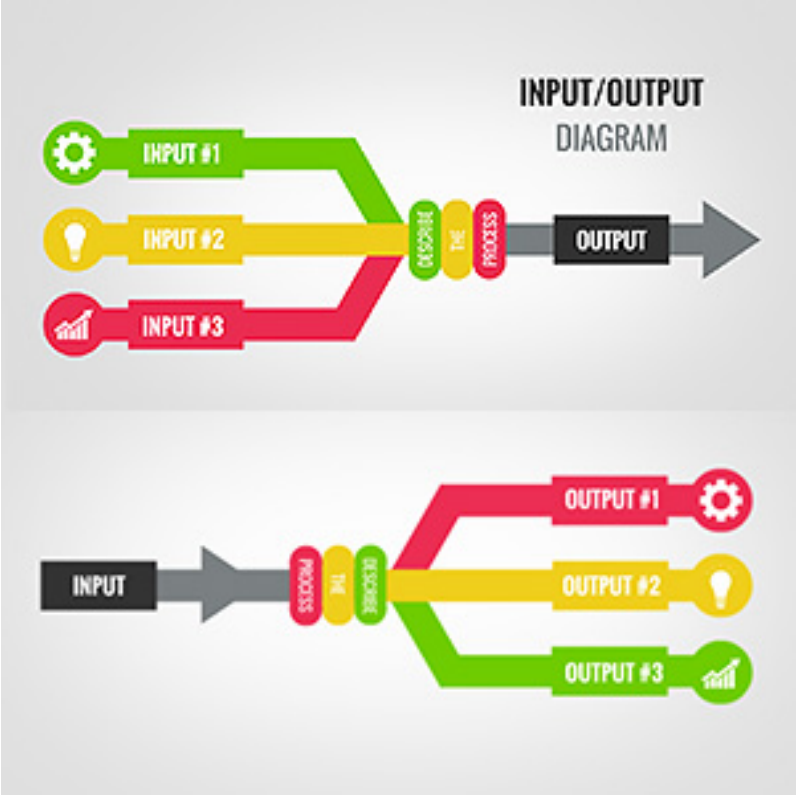

Figura 3.1: Diagrama *Input/Output*

Para a elaboração deste programa, as linguagens de programação utilizadas foram, na sua quase totalidade, *Python*, no desenvolvimento do cálculo dos elementos, e uma pequena percentagem de *Matlab*, no que diz respeito à apresentação de resultados. A primeira linguagem foi escrita com a utilização do editor de código *Visual Studio Code* integrado num dos mais usados e conceituados ambientes de Pyhton, *Anaconda*, que permite a utilização de um grande número de módulos matemáticos ideais ao cálculo deste método, tornando esta linguagem uma ótima escolha devido à sua versatilidade e dinâmica. A segunda linguagem foi escrita com recurso ao próprio programa de *Matlab* já conhecido dos estudantes de engenharia civil desde o 1º ano.

3.1 *Input*

Para quem não está familiarizado com o conceito, o chamado *Input* é uma expressão de origem inglesa que, traduzindo, significa entrada. Este termo é vastamente utilizado na sociedade e em muitas áreas da atividade humana, mas é bem mais usado e específico das áreas da engenharia, especialmente da engenharia informática e das áreas da tecnologia da informação.

De um modo geral, o *Input* é então um termo associado a qualquer tipo de informação que é enviada a um computador para que este a consiga processar. Sem a entrada de dados, um programa não consegue correr. Esta entrada pode ser efetuada de diferentes maneiras e com diferentes tipos

de dados, desde que o programa esteja apto para os receber.

Falando agora especificamente do *Input* do programa em causa, este tem de estar preparado e orientado de uma dada forma para que a quantidade e qualidade de dados que entrarão no programa possa crescer e ser melhorada com o decorrer do trabalho efetuado no mesmo. Numa fase inicial, toda a informação necessária para o cálculo de um elemento qualquer será compilada num ficheiro *JSON*.

3.1.1 *JSON para a especificação da entrada base do sistema*

JSON é um acrónimo de *JavaScript Object Notation*, que significa notação de objetos de *Javascript*, e é essencialmente um formato leve de troca de informações ou dados entre sistemas. Trata-se de um formato de serialização de dados, tal como XML por exemplo, no entanto, a grande diferença para esta alternativa e o que faz destacar este formato é a facilidade com que este pode ser lido por humanos e também por computadores. Devido à sua capacidade de estruturar a informação de uma forma bem mais compacta do que a conseguida pelo modelo XML, o *parsing*(análise) de toda a informação no formato *JSON* torna-se consideravelmente mais rápido.

Um formato de serialização é o que permite a transmissão de um objeto mais ou menos complexo, seja ele um dicionário composto por vários valores, *arrays* e *strings* ou somente um número inteiro, por um transporte simples e não-tipado.

Independentemente do tipo de linguagem que se esteja a usar no projeto, o formato-texto *JSON* é independente de qualquer tipo de linguagem e usa algumas convenções que são bastante familiares para utilizadores de linguagens como C++, C, *Java, JavaScript, Perl, Python* e muitas outras. Todas estas características fazem com que o *JSON* seja o formato de eleição para a troca e transmissão de dados entre sistemas e que atualmente seja utilizado por milhares de pessoas e por grandes empresas de renome como a *Google* e a *Yahoo*, por exemplo [8].

3.1.1.1 Sintaxe do *JSON*

A maneira como um *JSON* pretende representar a informação é do mais simples que pode haver. Para cada valor apresentado e que à partida se queira obter está associado um nome ou rótulo que classifica esse mesmo valor, ou seja, descreve o seu significado.

Um *JSON* é fundamentalmente construído por duas estruturas:

- Uma coleção de pares nome/valor. Estes pares têm designações diferentes de linguagem para linguagem e podem ser tratados como objetos, estruturas, dicionários e muitas outras designações, dependendo do caso em questão.

- Uma lista ordenada de valores. Neste caso, esta organização de valores pode ser conhecida como *array*, lista, vetor ou sequência, dependendo também da linguagem usada.

Estas estruturas de dados apresentadas são universais e transversais a qualquer linguagem. Hoje em dia, literalmente todas as linguagens modernas são capazes de interpretar este tipo de dados e estrutura associada. Não faria qualquer tipo de sentido se assim não fosse visto que estamos a lidar com um ficheiro que pode executar a transferência de dados entre diferentes linguagens e ficheiros. É imperativo que este formato seja de certa forma neutro.

Portanto, no formato *JSON*, a estrutura de dados a adotar é a seguinte:

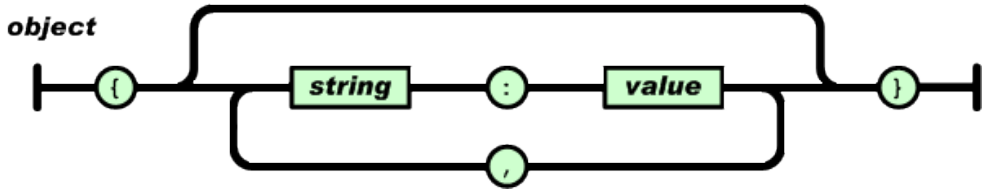

Figura 3.2: Objeto

Um objeto é simplesmente uma sequência de pares nome/valor não necessariamente ordenados por uma ordem específica. Para quem escreve em python, como é o caso deste trabalho, um objeto corresponde a um dicionário em que os nomes em causa são chamados de *keys*(chaves) que assumirão a forma de *string*. Um objeto é sempre delimitado por chavetas "{}"em que os nomes incluídos no mesmo são sempre seguidos por dois pontos ":"e de seguida o valor associado. Os pares nome/valor têm de ser separados por uma vírgula ",".

De seguida são dados três exemplo de objetos válidos:

1. {}
2. {“alfa”: 1}
3. {“alfa”: 1, “beta”: 2, “gama”: “três”, “delta”: {}}

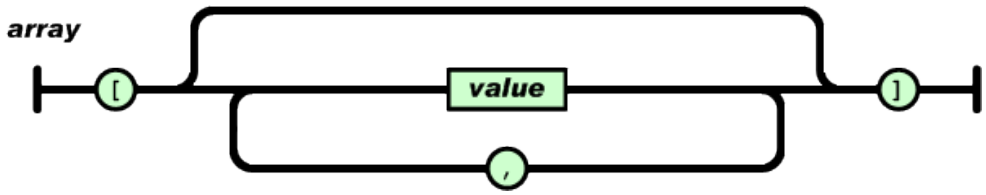

Figura 3.3: *Array*

Um *array* é um conjunto de valores ordenados de uma maneira específica. Em python, um array pode ser também chamado de lista e nele podem existir valores numéricos, *strings* ou até mesmo objetos. Este é delimitado por parênteses retos "[]"e o conteúdo nele inserido é separado por vírgulas ",".

Exemplos de *arrays* válidos:

1. []

2. [1]

3. [1,2,3]

4. [“um”, “dois”, “três”]

5. [1, “dois”, “um”: 1, “três”, 4, [5, 6, “sete”]]

No que diz respeito ao valor associado a cada nome do objeto, este pode assumir vários formatos. Como a figura indica, os valores podem aparecer na forma de *string* usando aspas para os delimitar, na forma de número, na forma de *boolean* que basicamente indica um valor verdadeiro (*True*) ou falso(*False*), pode aparecer como sendo um valor nulo(*null*) ou ainda ser um objeto ou array. É importante de perceber que dentro de um array podemos ter objetos ou até outros arrays, por exemplo.

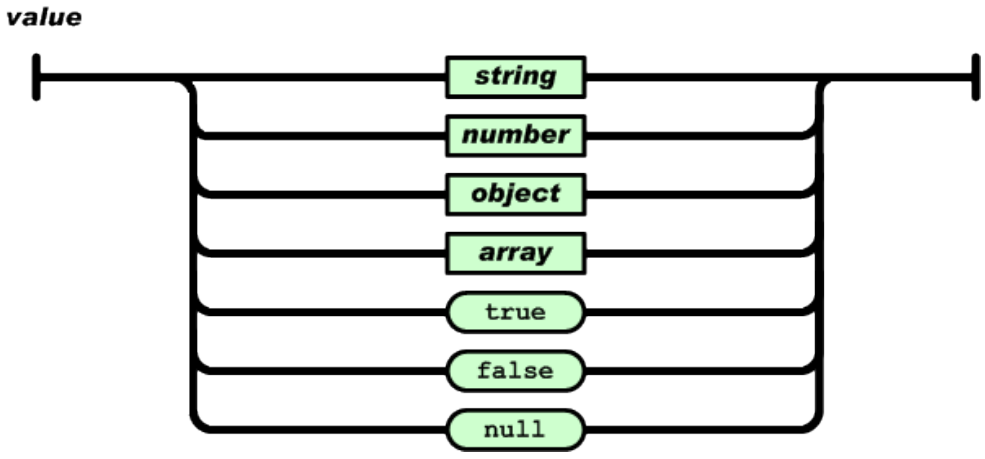

Figura 3.4: *Value*

Uma *string* é uma sequência de zero ou mais caracteres *Unicode*, delimitados por aspas em que os caracteres de escape são definidos por uma barra invertida (*backslash*). Uma string em

JSON é exatamente igual a uma *string* em *Python*, C ou *Java*.

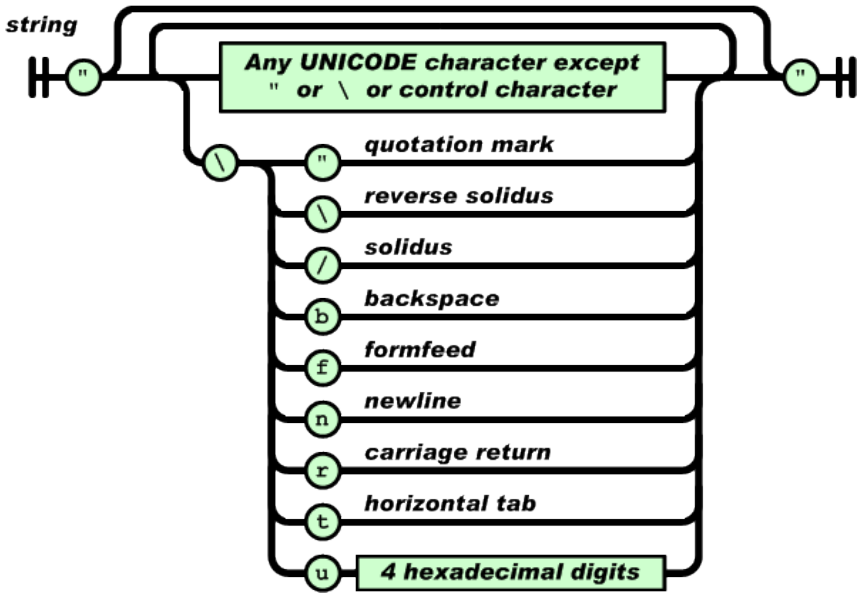

Figura 3.5: *String*

Um número, tal como noutras linguagens, pode ser inteiro ou não e é constituído por dígitos de 0 a 9

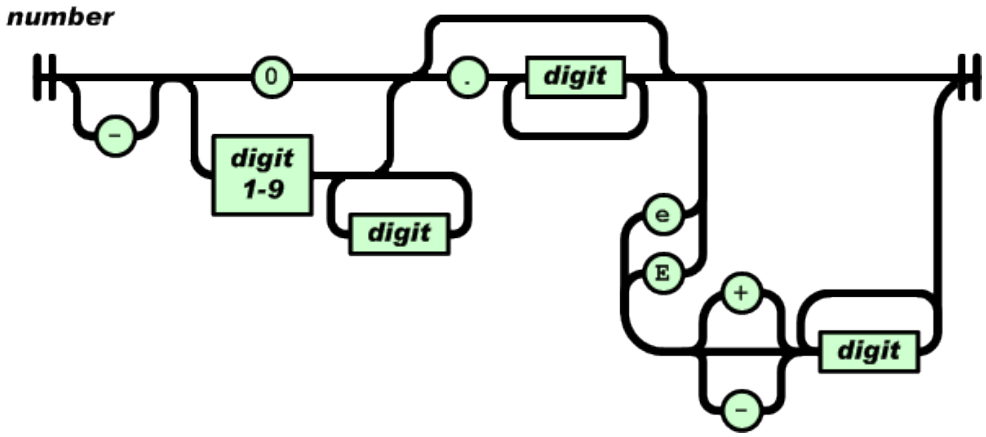

Figura 3.6: *Number*

Fora alguns detalhes de codificação que possam existir e mais alguns pequenos pontos relativos à estruturação do *JSON*, a descrição e uso da linguagem está praticamente feita.

Concluindo, usar um ficheiro *JSON* neste projeto é uma grande mais valia pois as suas características fazem dele uma das melhores opções da atualidade quando se está a lidar com grandes quantidades de informação que têm de ser acedidas e analisadas. As suas particularidades mais

relevantes pairam sobre uma leitura extremamente fácil e simples, um *parsing*(análise) muito acessível, o suporte de objetos, uma grande velocidade na execução e transporte de dados e ser um texto com tamanho reduzido.

Tendo então todo o conhecimento necessário para a compreensão e utilização de um *JSON*, a elaboração do mesmo para este projeto é de extrema facilidade.

3.1.2 Transversalidade do *Input*

Este trabalho, como já referido anteriormente, tem como objetivo o desenvolvimento de um programa de elementos finitos em ambiente aberto. Com isto, é importante compreender que este conceito implica que todo o trabalho desenvolvido em prol deste mesmo programa não se limita a esta dissertação, mas sim também, a todas as contribuições que possam existir no futuro. A elaboração de um *software* desta categoria, com a qualidade que lhe é exigida, é de extrema dificuldade e tem uma duração de anos até que o produto esteja completamente apto e aceitável para qualquer tipo de análise. Nesta fase inicial, e desenvolvido nesta dissertação, a obtenção de resultados é referente a uma análise estática e linear de elementos quadriláteros, de quatro nós, no estado plano de tensão. Toda a entrada de dados consistirá, utilizando um *JSON*, na introdução das coordenadas dos diferentes elementos, do material e características geométricas a estes associado, e de todas as condições a que se encontra submetido, respeitantes às cargas aplicadas, assentamentos e apoios. É de esperar que futuramente, alunos da especialidade de estruturas, e não só, possam contribuir não só com novos elementos, mas também neste caso, com novos tipos de *input* para que posteriormente se consiga ter um programa com uma maior diversidade, visto que este foi elaborado com esse objetivo. Atualmente existem dezenas de entradas diferentes que podem complementar este programa, a vantagem de conseguir preparar o projeto para novas entradas é que depois basta receber novas contribuições e inseri-las naturalmente.

3.1.2.1 *Graphical User Interface*

Uma *Graphical User Interface*(GUI) é um tipo de interface do utilizador que permite aos mesmos navegar num computador ou dispositivo e completar todo o tipo de ações através de indicadores visuais e ícones gráficos.

Todos os grandes sistemas operativos da atualidade como o *Windows*, o *Mac*, *iOS* e *Android* servem-se de uma interface gráfica onde é possível clicar num ícone para completar uma ação como por exemplo abrir uma aplicação ou programa, vêr um menu ou navegar através de um dispositivo.

Contrariamente a uma *Command Line Interface*(CLI), uma GUI torna-se consideravelmente mais fácil de se usar e aprender por parte de quem é principiante nesta matéria. A vantagem de usar este tipo de interface é que não existe a necessidade de memorizar comandos e os utilizadores não precisam de qualquer tipo de conhecimento de linguagens de programação. Estas tornam a

interação com o utilizador o mais simples possível desde a abertura de menus, o movimento de ficheiros ou o normal correr de um programa, sem que seja preciso dizer ao computador o que fazer através da linha de comandos.

Com a facilidade de entrada de dados e a neutralidade que um *JSON* oferece, a inclusão de, por exemplo, uma *graphical user interface*(GUI), ou seja, uma interface onde o utilizador introduz os dados necessários ao cálculo do elemento de uma maneira mais prática, não exige grande dificuldade. Os valores são introduzidos na interface pelo utilizador onde de seguida, estes são mapeados para o formato *JSON*, encaixando na perfeição neste projeto, sem que seja preciso rescrever código e fazer qualquer tipo de alterações. Portanto, no uso de uma interface gráfica, valores referentes à geometria e ao material em questão são exemplos de introdução de dados para que depois possam ser organizados para o formato *JSON*.

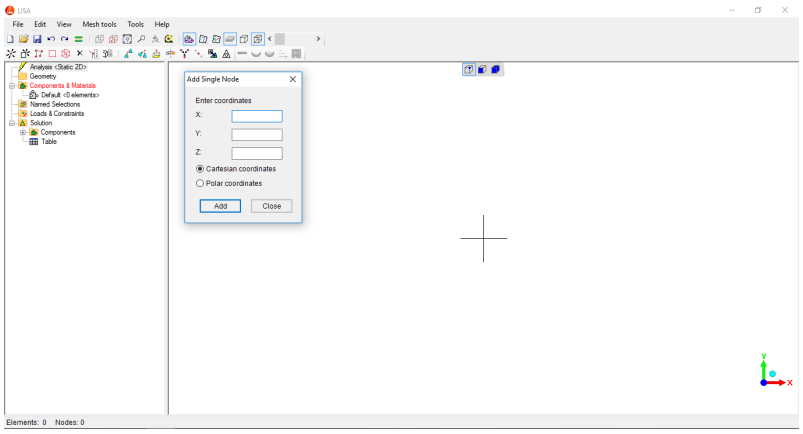

Figura 3.7: Exemplo de introdução de coordenadas numa GUI

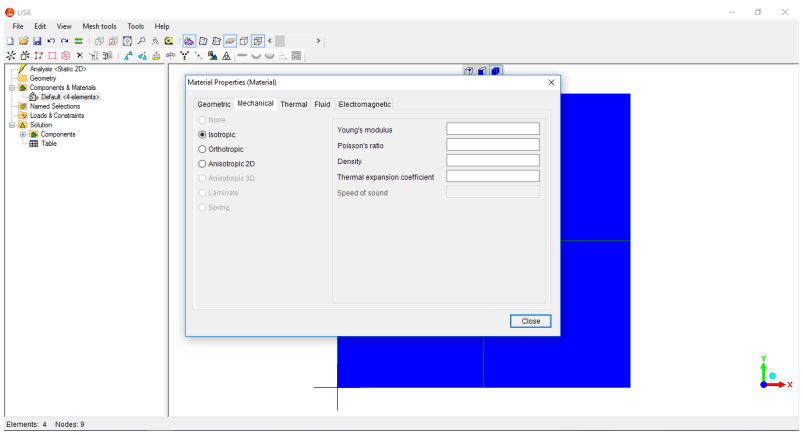

Figura 3.8: Exemplo de introdução das características do material numa GUI

3.1.2.2 *Comma-Separated Values*

O formato de um ficheiro *Comma-Separated Values*(CSV) é um formato simples de armazenamento de dados que agrupa as informações de arquivos de texto em folhas de cálculo. Este tipo de ficheiros pode ser facilmente criado e editado recorrendo ao *Excel*.

Como o próprio nome indica, um ficheiro deste género não tem formatação e os valores que o constituem estão separados por vírgulas, delimitados por aspas e cada linha tem um registo diferente.

Vejamos o seguinte exemplo de uma linha em CSV :

"Pêra","Origem: Portugal","871","5412","1,10","1","10"

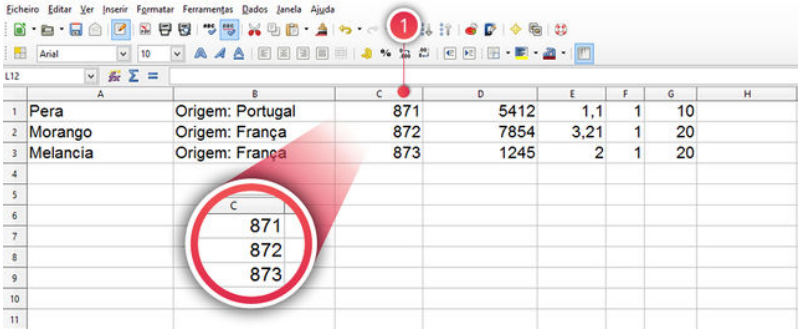

Figura 3.9: Exemplo de um ficheiro csv numa folha de cálculo

Da mesma maneira que é possível inserir uma interface gráfica, outro tipo de entrada de dados também é facilmente utilizado, bastando apenas o mapeamento ou organização destes mesmos para uma estruturação correta. Se a informação dos dados estiver num formato *comma-separated values* (CSV) ou até mesmo num arquivo de texto(TXT), por exemplo, estes terão apenas de ser copiados e transformados no formato necessário para correr o programa, neste caso, um *JSON*.

3.1.2.3 *Drawing Interchange Format*

Um outro tipo de entrada que poderia ser interessante de obter também é a importação de ficheiros do tipo *Drawing Interchange Format*(DXF). Este formato de ficheiros foi desenvolvido pela *Autodesk* e consiste em desenhos do tipo *CAD*. Sendo o *Autocad* uma ferramenta muito usada por engenheiros civis e muito boa para o desenho de estruturas, a possibilidade de importar este tipo de ficheiros para o programa para que depois possam ser resolvidos é uma boa adição a este projeto. Esta ferramenta é de fácil e rápida aprendizagem e vantajosa se se estiver a trabalhar num projeto específico e for de interesse saber as tensões e deslocamentos associados à estrutura em causa, seja ela uma viga, um pilar ou um pórtico, em duas ou três dimensões. Esta abordagem é

propícia quando se quer analisar grandes estruturas em que a sua geometria está já definida neste formato. Se não for o caso, acaba por se tornar algo trabalhosa a sua definição.

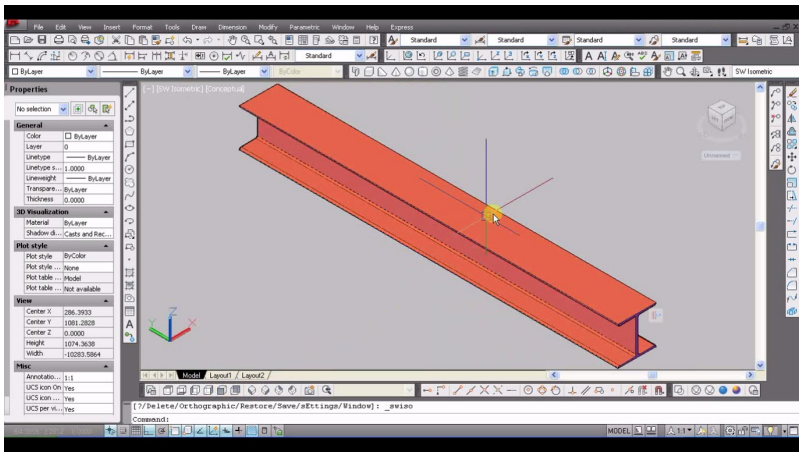

Figura 3.10: Exemplo de uma possível entrada em DXF

Depois de importado o ficheiro no formato DXF, é só preciso definir o material a este associado, as suas respetivas características e as condições em que este se encontra para que a sua análise seja possível.

3.1.3 Conteúdo do ficheiro *JSON* de entrada

Nesta primeira etapa, toda a entrada de informações será feita e armazenada através de um ficheiro *JSON*. Este ficheiro organiza toda a informação que vai entrar no programa e é importante que esta seja escrita de uma maneira correta para que o cálculo seja bem sucedido. Para tal, foram criadas funções de validação de *input* para garantir que o programa reconhece quando algum valor não é bem introduzido ou se a informação em causa foi escrita de maneira errada. Se tal acontecer, o programa simplesmente não correrá e exibirá uma mensagem ao utilizador do tipo de erro em questão. A informação presente está então organizada pelos valores e dados imprescindíveis ao cálculo do respetivo elemento, valores estes correspondentes às coordenadas, aos materiais e às condições em que o elemento se encontra.

De seguida são apresentadas as várias partes deste ficheiro de entrada, explicada a maneira como este deve ser construído e a forma como estes dados são integrados no programa em causa.

3.1.3.1 Coordenadas

O primeiro passo na formulação de um problema de elementos finitos é a caracterização das coordenadas presentes para que estas possam mais tarde definir um elemento tipo. Assim sendo, as coordenadas a usar são predispostas num *array* em que cada entrada deste mesmo corresponde a um ponto e às suas respetivas coordenadas em x, y e z. Nesta fase inicial, só se lidou com

elementos calculados no estado plano de tensão, portanto, o código de seguida apresentado apenas apresenta coordenadas em 2 direções. Se porventura se estivesse a lidar com um problema de 3 dimensões, bastaria inserir mais uma entrada no *array* dos pontos em causa.

```
"coord_a": [

  [ "P1",   0,  0 ],
  [ "P2",   2,  0 ],
  [ "P3",   4,  0 ],
  [ "P4",   0,  2 ],
  [ "P5",   2,  2 ],
  [ "P6",   4,  2 ],
  [ "P7",   0,  4 ],
  [ "P8",   2,  4 ]

 ]
```

Listing 3.1: Exemplo de um array de coordenadas no ficheiro JSON

3.1.3.2 Elementos

Com os pontos já discriminados no plano, são formados então os diferentes tipos de elementos que serão alvo de cálculo no programa. O objeto apresentado de seguida contém a informação referente a cada elemento dos vários tipos de elementos que se possa usufruir. Cada um destes é inserido num *array* respetivo ao seu tipo com os dados necessários para que este possa ser devidamente formado. Tal como os pontos previamente definidos, cada elemento terá a sua numeração, designada de *name*. De seguida, o material em questão terá de ser definido também. Cada tipo de material terá uma nomenclatura definida para que o programa consiga interpretar e ir buscar a informação necessária, respetiva a esse mesmo material. Adotou-se a designação presente no Eurocódigo 2 para a definição das características materiais do betão. O mesmo poderá ser feito com qualquer tipo de material a usar. Um dos diretórios presentes no programa, chamado de *materials*, armazena todos os dados característicos de um dado material. Posteriormente, surge a formação dos elementos em causa. O exemplo dado, *plane/EL_stress_4N_4E*, referente a um elemento de 4 lados e 4 nós no estado plano de tensão, terá de conter um *array* de coordenadas com 4 pontos, correspondentes aos seus 4 nós. A ordenação destes pontos terá de ser efetuada segundo a convenção usualmente utilizada, no sentido anti horário, para que se possa facilmente calcular cada elemento no seu referencial local. Por último, é contemplada a espessura de cada nó do respetivo elemento, uma característica dos problemas do estado plano de tensão. Se o array que compõe esta informação, *height_a*, for formado apenas por um único valor, o programa assumirá que todos os nós terão o mesmo valor, caso contrário, terá de ser definida a espessura para cada nó, ficando neste caso um array com 4 entradas.

```
"plane/EL_stress_4N_4E":[
    {
        "name":"E1",
        "material":"concrete/C25",
        "coord_a":[
            "P1",
            "P2",
            "P5",
            "P4"
        ],
        "height_a":[
            0.3
        ] },
    {
        "name":"E2",
        "material":"concrete/C25",
        "coord_a":[
            "P2",
            "P3",
            "P6",
            "P5"
        ],
        "height_a":[
            0.3
        ]  },
    {
        "name":"E3",
        "material":"concrete/C25",
        "coord_a":[
            "P4",
            "P5",
            "P8",
            "P7"
        ],
        "height_a":[
            0.3
        ]
    }
]
```

Listing 3.2: Exemplo de objeto dos diferentes tipos de elementos de entrada no programa

3.1.3.3 Ligações ao exterior

Com os elementos totalmente caracterizados, são impostas agora as ligações ao exterior da estrutura a ser calculada. Uma vez mais, como os elementos calculados nesta etapa dizem respeito ao estado plano de tensão, o número de graus de liberdade em cada nó vai ser igual a 2, portanto, os apoios de seguida demonstrados só poderão restringir deslocamentos nas direções x e y. Do mesmo modo que nas coordenadas, quando posteriormente se lidar com problemas com mais graus de liberdade por nó, basta apenas inserir mais entradas no *array* correspondente.

Para formalizar as ligações ao exterior é necessário definir primeiro o ponto em causa e posteriormente definir as condições a que este estará submetido. Estas condições são definidas num array de 2 colunas, correspondentes aos graus de liberdade presentes. Se a respetiva direção estiver travada, ou seja, o deslocamento for zero, deverá constar a letra "F". Se a direção estiver livre, isto é, o deslocamento estiver por determinar, o valor correspondente a essa direção no array é 0 (zero). Finalmente, se por acaso existir um assentamento de apoio, o valor deste terá de ser introduzido na direção respetiva também.

```
"fixed_point_a": [
    { "point": "P1", "disp_a": [ "F", "F" ] },
    { "point": "P3", "disp_a": [  0 , "F" ] },
    { "point": "P4", "disp_a": [ 0.01 , 0 ] }
  ]
```

Listing 3.3: Exemplo do um array das diferentes ligações ao exterior

3.1.3.4 Casos de carga

Qualquer estrutura estará submetida a diferentes tipos de ações e é do interesse de qualquer pessoa que recorra a um programa de elementos finitos estudar a maneira como estas influenciam o comportamento da estrutura. É muito raro ter todo o tipo de ações a atuar ao mesmo tempo, mas é importante serem efetuadas as diferentes combinações destas mesmas ações para que seja mais percetível o tipo de tensões que mais frequentemente estarão presentes e em que zonas. Portanto, é apresentado de seguida o exemplo dos diferentes casos de carga e, dentro de cada caso de carga, os diferentes tipos de carga, sejam elas cargas nodais concentradas ou cargas distribuídas.

Para definir uma carga concentrada é necessário apenas especificar o ponto em que esta se encontra aplicada e o seu valor. O *array* correspondente ao valor, *value_a*, é um *array* com um número de entradas correspondente ao número de dimensões do problema, ou seja, a direção onde a carga está aplicada.

No que toca à aplicação de cargas distribuídas, estas terão de ser dispostas nas faces dos elementos em causa. O *edge_a*, é um *array* que contém os pontos onde a carga estará aplicada e que possui tantas entradas quanto o número de nós em cada face do elemento em questão. O valor corresponde à carga aplicada ao longo dessa face é transcrito num *array*, *value_a*, também ele

com o mesmo número de entradas do anterior e que, em cada uma dessas entradas, estará o valor correspondente desse nó na direção x, y ou z. É importante frisar que os valores deste tipo de carga estão representados num referencial local. Um exemplo desta aplicação, para cargas horizontais que assumam um valor no sentido da esquerda para a direita, é observável no caso de carga *wind*, onde valor inserido é de -20 da direção y do referencial local.

Definidos todos os casos de carga presentes no problema, surge de seguida e por último no ficheiro de *input* a combinação destes mesmos. Para cada combinação é definido um *array* com o número de casos de carga definidos anteriormente em que, cada entrada corresponde ao peso do mesmo no problema. Assim sendo, se porventura se pretender analisar o comportamento da estrutura submetida à totalidade do seu peso próprio, mas apenas a metade das ações devidas ao vento, como exemplificadas na lista, é necessário inserir os valores de 1 e 0.5 neste mesmo *array* de combinações.

```
{
    "loads_a":[
        {
            "name":"dead load",
            "concentrated":[
                {
                    "point":"P6",
                    "value_a":[
                        10,
                        0
                    ]
                }
            ],
            "edge_distributed":[
                {
                    "element":"E1",
                    "edge_a":[
                        "P1",
                        "P4"
                    ],
                    "value_a":[
                        [
                            0,
                            0
                        ],
                        [
                            0,
```

```
                    0
                ]
            ]
        }
    ]
},
{
    "name":"wind",
    "concentrated":[
        {
            "point":"P3",
            "value_a":[
                0,
                0
            ]
        }
    ],
    "edge_distributed":[
        {
            "element":"E1",
            "edge_a":[
                "P1",
                "P4"
            ],
            "value_a":[
                [
                    0,
                    -20
                ],
                [
                    0,
                    -20
                ]
            ]
        }
    ]
}
],
"combinations":[
    {
```

```
            "name":"Combination 1",
            "w":[
                1,
                0.5
            ]
        },
        {
            "name":"Combination 2",
            "w":[
                1.5,
                0.7
            ]
        },
        {
            "name":"Combination 2",
            "w":[
                1.2,
                0
            ]
        }
    ]
}
```

Listing 3.4: Exemplo dos diferentes tipos de casos de carga e as suas combinações

À medida que o programa vai ficando cada vez mais completo, isto é, com a adição de novos tipos de elemento, novas condições de apoio e cargas, este ficheiro terá também novas entradas para que seja possível complementar o código correspondente ao cálculo destas novas características.

3.2 Cálculos

Esta segunda parte do programa contempla a zona de código que é responsável pelo processamento da informação recolhida na zona de entrada, produzindo desse modo os resultados que serão apresentados ao utilizador sob uma forma de ficheiro ou de um modo gráfico. Neste contexto, quando se fala de cálculos, é implícito que esta é a fase em que se recorre ao método dos elementos finitos para toda a computação necessária para a análise desejada.

Tal como qualquer outra parte do programa, a estruturação desta não é exceção. Como foi observado anteriormente na fase do *Input*, a transversalidade requerida é de grande nível e se tudo estiver bem organizado, as novas entradas serão inseridas sem grandes dificuldades. Quando se exprime o desejo de novas entradas nesta fase, é legítimo afirmar que estas poderão ser de

diferentes géneros, isto é, tanto pode ser informação relativa a um elemento ou até mesmo novos métodos de resolução do sistema de equações utilizado no cálculo dos respetivos elementos.

Quando se está a lidar com um software de elementos finitos, e até mesmo outro programa qualquer, a disposição dos ficheiros que o constituem é quase sempre a mesma. Para uma fácil interpretação de quem os usa e de quem contribui, estes ficheiros são dispostos em diretórios e sub-diretórios, que dizem respeito às diferentes fases de cálculo do programa.

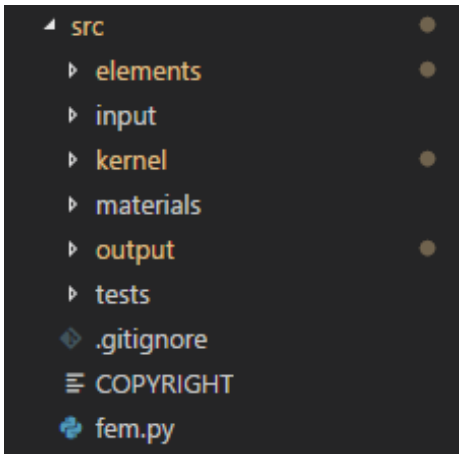

Figura 3.11: Exemplo de estruturação de um programa de elementos finitos

Como é constatável na figura anteriormente apresentada, o *source code* do programa é constituído apenas por alguns diretórios essenciais ao cálculo de qualquer elemento finito e esta será sempre a sua estrutura base, independentemente do número de entradas que possam surgir. A acompanhar todos os diretórios necessários surge sempre um *main.py*, neste caso chamado de *fem.py*, que corresponde ao ficheiro principal do programa. É neste ficheiro onde está presente o esqueleto do programa em si e onde são chamadas todas as funções necessárias para a executar o mesmo, sem que seja preciso estar constantemente a fazer alterações. Este é um dos pontos principais que permite a entrada natural de código no programa.

3.2.1 Breve noção do Método dos Elementos Finitos

Para proceder ao cálculo e elaboração de um programa de elementos finitos, é preciso primeiro perceber de uma maneira correta e prática como é que este funciona. O método em questão é a génese do funcionamento e estruturação do programa elaborado.

No cálculo que qualquer estrutura, seja ela de uma, duas ou três dimensões, é necessário definir a geometria da mesma. Esta geometria, submetida a diferentes condições de carga, apoios e deslocamentos, é subdividida em pequenas parcelas, denominadas de elementos, que passam então a definir o domínio contínuo do problema. A subdivisão da geometria inicial em fragmentos mais pequenos permite a resolução de um problema à partida complexo, agora dividido em problemas mais simples. Este método de cálculo é facilmente resolvível por um computador. O que o

método preconiza então é que um número infinito de variáveis desconhecidas seja substituído por um número finito de elementos com um comportamento entendível. As subdivisões, chamadas de elementos, podem assumir as mais variadas formas, desde triângulos, quadriláteros ou até elementos unidimensionais. O uso destes varia com o tipo e dimensão do problema e cada elemento é calculado de uma maneira distinta.

Quando subdivididos sobre o domínio da estrutura, os elementos em questão são ligados entre si por pontos, que no método possuem a denominação de nós ou pontos nodais. Quando se procede ao agrupamento de todas essas parcelas, elementos e nós, no domínio a analisar, surge então conjunto de elementos denomidado por malha. Como o método dos elementos finitos é um método numérico, as equações matemáticas que definem o comportamento físico de cada elemento não serão resolvidas de maneira exata, mas sim de forma aproximada. Portanto, a quantidade de nós e elementos utilizados num dado problema, o seu tamanho e o tipo de malha presente vão ser determinantes na precisão do método. Quanto menor o tamanho e maior o número de elementos numa determinada malha, maior será a precisão dos resultados obtidos.

Concluindo, quanto maior o número de elementos presentes num software maior será a capacidade do mesmo de proceder ao cálculo de estruturas mais complexas e com uma maior precisão. Um conhecimento profundo do método é fundamental para que o programa a desenvolver seja eficiente. A perceção de como este funciona e como se procede ao cálculo dos diferentes elementos singularmente e à assemblagem dos mesmos é que vai permitir a inclusão de novas componentes no projeto.

3.2.2 *Elements*

Um dos diretórios incluídos na parte do *source code* é o diretório dos *elements*. Neste diretório estão presentes todos os tipos de elementos que podem ser incluídos no programa. Mediante o tipo de problema em questão, é necessário dividir este diretório em sub-diretórios correspondes o modelo do elemento pretendido. Na figura apresentada de seguida pode-se observar a existência do sub-diretório *plane* que conterá toda a informação específica de um dado elemento no estado plano, seja ele de tensão ou deformação. Posteriormente, outros diretórios podem ser adicionados, correspondendo a elementos do tipo *shell*, axissimétricos ou sólidos, por exemplo.

Dentro de cada diretório correspondente a um elemento em específico, existem todos os ficheiros necessários para a resolução do mesmo para que, na altura de proceder ao cálculo, toda a informação que não seja genérica e que seja característica de um dado tipo de elemento possa ser encontrada e usada facilmente, criando-se assim um programa claro e fácil de ser entendido.

Como todos os elementos terão o mesmo tipo de informação, tome-se o exemplo da figura 3.12, do tipo de elemento *EL_stress_4N_4E*. Este conterá ficheiros com dados específicos do cálculo deste elemento que será necessário nas diferentes fases do programa.

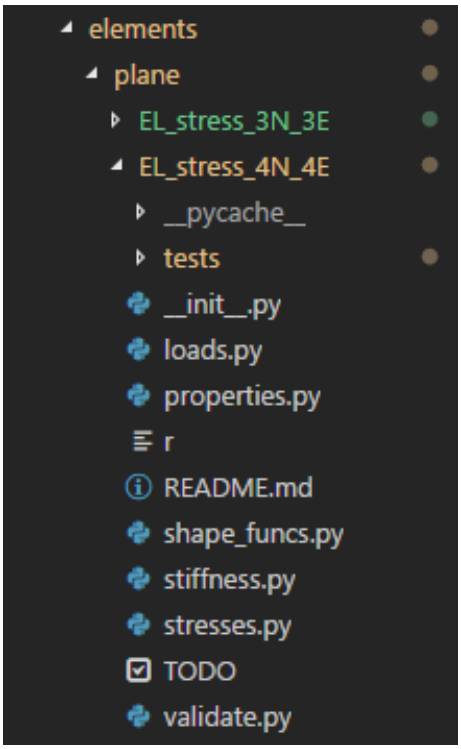

Figura 3.12: Diretório *elements*

3.2.2.1 *Properties*

Falando do ficheiro *properties*, cada elemento terá a sua especificidade e características físicas inerentes [14]. Para que o programa possa ser escrito de uma maneira limpa e genérica, cada um destes ficheiros conterá um objeto com todas as características necessárias previamente definidas.

```
data = {
 "nnode": 4,
 "nedge": 4,
 "nstre": 3,
 "ntype": 1,
 "ndime": 2,
 "ndofn": 2,
 "dof_l": [ "Displacement_x", "Displacement_y" ]
};
```

Listing 3.5: Exemplo das propriedades de um elemento tipo

3.2.2.2 *Shape functions*

No ficheiro *shape_funcs* está presente a informação relativa às funções de forma de um elemento. Como se sabe, cada elemento vai possuir as suas funções de forma que são já conhecidas e que podem ser previamente escritas. Quando procedemos ao cálculo de um elemento finito e procedemos à análise deste num referencial local, estas funções tornam-se imutáveis.

```
## Local coords
s1 = Symbol("s1")
s2 = Symbol("s2")

si_a = [ s1, s2 ];

## Shape functions
N1 = -(s1/4)-(s2/4)+((s1*s2)/4)+1/4;
N2 =  (s1/4)-(s2/4)-((s1*s2)/4)+1/4;
N3 =  (s1/4)+(s2/4)+((s1*s2)/4)+1/4;
N4 = -(s1/4)+(s2/4)-((s1*s2)/4)+1/4;

shape_func_list = [ N1, N2, N3, N4 ];
```

Listing 3.6: Exemplo de funções de forma de um elemento de 4 nós e 4 faces

3.2.2.3 *Stiffness*

Avançando para o ficheiro *stiffness*, este irá possuir as funções com especificidades do elemento necessárias para o cálculo da sua matriz de rigidez. O facto de estarmos a lidar com diferentes elementos em diferentes tipos de problemas, implica que as matrizes de deformação e elasticidade estarão em constante mutação, seja em conteúdo e em dimensão, portanto, estas têm de ser definidas de acordo com o problema e elemento pretendido e posteriormente chamadas quando forem requeridas. De seguida é apresentada a forma como é construída a matriz de deformação sendo que, para tal, é necessário conhecer as derivadas das respetivas funções de forma, denominadas de *derivatives_matrix*.

```
def build_deformation_matrix( shape_func_list,
    derivatives_matrix ):

  B = np.zeros((3,0),dtype=Symbol);

  for i in range (len(shape_func_list)):
    b=np.array([[derivatives_matrix[i,0], 0 ],
```

```
                [0, derivatives_matrix[i,1]],
                [derivatives_matrix[i,1], derivatives_matrix[i
                    ,0]]]);

    B = np.concatenate((B,b),axis=-1)

  return B;
```

Listing 3.7: Exemplo da construção de uma matriz de deformação

3.2.2.4 *Loads*

Como referido anteriormente, o cálculo das forças nodais equivalentes referente a uma carga distribuída depende sempre do tipo de elemento em causa. No ficheiro *loads* encontra-se a informação específica para a determinação das forças quando consideramos um referencial local na face do elemento onde a carga estará aplicada.

```
def data():

  s = Symbol("s")

  si_a = [ s ];

  ## Shape functions
  N1 = (1-s)/2;
  N2 = (s+1)/2;

  shape_func_list = [ N1, N2 ];

  ##
  used_gauss_points = [ 1 ];

  return {
    "load_types_a": [ "edge_distributed" ],
    "si_a": si_a,
    "shape_func_list": shape_func_list,
    "used_gauss_points": used_gauss_points
  };
```

Listing 3.8: Informação necessária ao cálculo das forças nodais equivalentes

3.2.2.5 *Stresses*

Por último, quando se procede ao cálculo do estado de tensão do elemento, é recorrente calcular as tensões nos pontos de Gauss e efetuar a sua extrapolação pois desta forma obtém-se resultados mais precisos do que aqueles que se obteriam com a avaliação direta das tensões no ponto pretendido [10].Portanto, no ficheiro *stresses*, vamos conter a informação necessária para o cálculo e extrapolação destas grandezas que dependerá de elemento para elemento [12].

```
def data( ):

    ## Gauss coordinates

    px=[-1/mt.sqrt(3),1/mt.sqrt(3),1/mt.sqrt(3),-1/mt.sqrt(3)]
    py=[-1/mt.sqrt(3),-1/mt.sqrt(3),1/mt.sqrt(3),1/mt.sqrt(3)]

    ## Interpolate coordinates

    pix=[-mt.sqrt(3),mt.sqrt(3),mt.sqrt(3),-mt.sqrt(3)]
    piy=[-mt.sqrt(3),-mt.sqrt(3),mt.sqrt(3),mt.sqrt(3)]

    return {
    "interpolate_coordinates" : [pix,piy],
    "gauss_coordinates": [px,py]
}
```

Listing 3.9: Informação necessária ao cálculo do estado de tensão de um elemento

3.2.3 *Materials*

O diretório *materials* terá incluída toda a informação referente ao material disponível para a atribuição ao elemento. Especificando no ficheiro *JSON* de entrada, qual o tipo de material que se irá usar para um dado elemento, a sua obtenção é de extrema facilidade pois as suas propriedades, tal como as propriedades de um elemento, estão guardadas na forma de um objeto, onde basta especificar a *key* da grandeza em questão, módulo de elasticidade ou coeficiente de *poisson* por exemplo, para se obter o valor pretendido.

```
def get():

  data = {
    "E": 31000000,
```

```
    "poisson_ratio": 0.3,
    "dt": 1e-5
};

return data;
```

Listing 3.10: Definição das propriedades de um material

3.2.4 *Kernel*

O diretório *kernel* é sem dúvida o diretório mais extenso e é onde se procede a todos os cálculos do método dos elementos finitos propriamente dito. É o núcleo central onde se executa todas as tarefas necessárias à discretização, assemblagem e demonstração de resultados de um ou mais elementos. Aqui todo o cálculo é genérico pois todas as características específicas de um dado elemento foram já definidas anteriormente. A somar ao cálculo em si é também executada a importação dos módulos com a informação especifica do tipo de elemento em questão, são estabelecidas as variáveis de ambiente necessárias à *performance* do programa e são definidas funções de *logging* que nos permitem registar o que vai acontecendo com o programa cada vez que este é corrido.

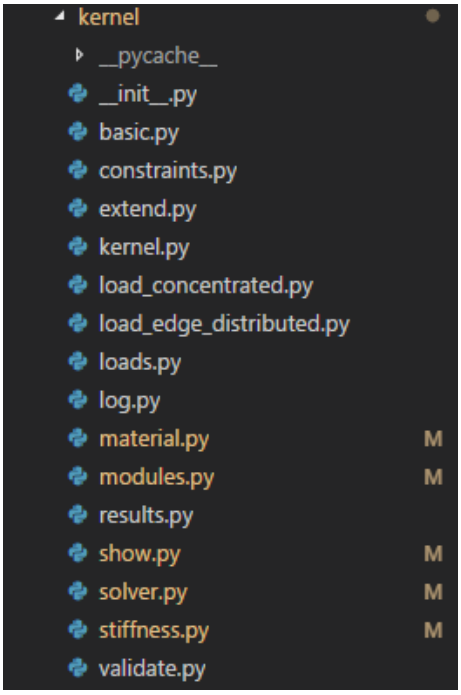

Figura 3.13: Diretório *kernel*

3.2.4.1 *Basic*

Para que o programa consiga ser executado em qualquer computador, todas as variáveis de ambiente têm de ser definidas previamente, portanto, neste ficheiro basic, é composta a função que atribui as variáveis necessárias já definidas no sistema operativo.

Uma variável de ambiente é um valor dinâmico, carregado na memória do computador, que pode ser utilizado por vários processos que funcionam simultaneamente. Na grande maioria dos sistemas operativos, o lugar de certas bibliotecas, ou mesmo os principais executáveis do sistema, podem ficar em lugares diferentes, dependendo da instalação. Portanto, graças às variáveis de ambiente, é possível, a partir de um programa, fazer a referência a um local tendo por base as variáveis de ambiente que definem estes dados.

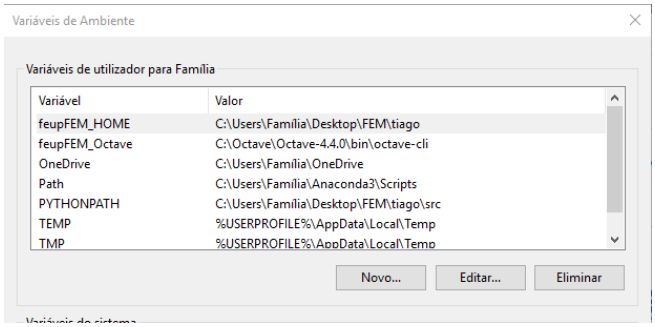

Figura 3.14: Definição de variáveis de ambiente no painel de controlo

Dentro deste ficheiro vamos ter então a função *init* que atribui às variáveis *feupFEM_HOME* e *feupFEM_Octave* o seu respetivo valor de variável de ambiente.

```
def init():
  global feupFEM_HOME;
  global feupFEM_Octave;
  try:
    feupFEM_HOME = os.environ[ 'feupFEM_HOME' ];
        except Exception as e:
    print ( "FATAL: environment variable feupFEM_HOME is not set
       ." )
    sys.exit ( 1 );

  try:
    feupFEM_Octave = os.environ[ 'feupFEM_Octave' ];
        except Exception as e:
    print ( "FATAL: environment variable
              feupFEM_Octave is not set." )
    sys.exit ( 1 );
```

```
return;
```

Listing 3.11: Função de atribuição de variáveis de ambiente

3.2.4.2 *Log*

Para que seja possível manter um registo de tudo que se passa no programa desde que este é corrido até à sua paragem, são definidas neste ficheiro funções de *logging* que escrevem e notificam todo o tipo de informações que passam no programa deste pequenos avisos, erros ou *debugs*.

```
def error(msg):
  logger.error(msg)

def info(msg):
  logger.info(msg)

def debug(msg):
  logger.debug(msg)
```

Listing 3.12: Algumas funções de *logging*

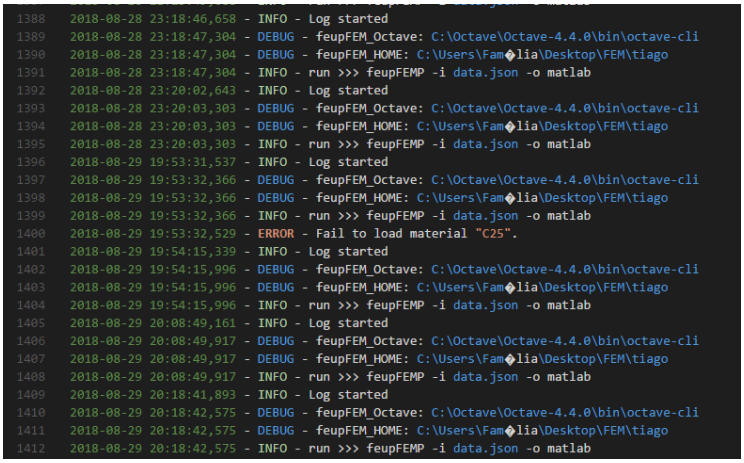

Figura 3.15: Informação de *logging*

3.2.4.3 *Extend*

Toda a informação necessária para a formulação e resolução do método dos elementos finitos é fornecida através do ficheiro *JSON* de *input*, no entanto, esta pode ser ligeiramente modificada de modo a facilitar a execução dos cálculos. Ao longo do programa surgem várias situações em que é preciso manipular toda a informação que se possui e transformá-la num formato adequado,

daí a existência deste ficheiro *extend*. É aqui que todos os dados iniciais, guardados na variável *data*, são estendidos e transformados para situações específicas para que posteriormente possam ser acedidos através desta mesma variável.

Um bom exemplo do uso do *extend* é para o mapeamento de valores de um referencial local para um referencial global. Quando as coordenadas de um ponto são definidas, a este ponto é definido uma numeração também que posteriormente definirá um elemento. A cada ponto corresponderá um certo número de graus de liberdade que indica precisamente a posição de um determinado valor de rigidez na matriz de rigidez global. Este é apenas um exemplo de informação que tem de obrigatoriamente ser estendida para que se possa proceder aos cálculos necessários.

```
point_dofns_h = {};
for i in points_h:
  g_point = points_h[i];
  kdofn = g_point * ndofn;
  dof_list = [];
  for idofn in range ( ndofn ):
    dof_list.append(kdofn);
    kdofn += 1;

  point_dofns_h[ i ] = dof_list;

data ["point_dofns_h"] = point_dofns_h;
```

Listing 3.13: Exemplo de criação de um objeto de mapeamento

3.2.4.4 *Kernel*

Quando se lida com um programa de elementos finitos é implícito que a quantidade de cálculos vai ser muito elevada, devido à quantidade de elementos que se tem de calcular. Basta atendermos à definição do método que permite o cálculo de apenas um elemento e a sua extensão e assemblagem de forma a cobrir um domínio desejado. Muitos cálculos implicam muita memória para o computador e a melhor maneira de tornar a resolução dos problemas mais eficiente é mesmo reduzir o número de cálculos que se sabe que à partida que se vão repetir. Para que não seja necessária a execução constante de ciclos por cada cálculo na resolução de um elemento, foi preparada uma função, chamada de *walk array* que, para cada elemento tipo, perfaz um ciclo por cima de todos os elementos. Assim sendo, quando chegar a altura de efetuar todos os cálculos necessários até à obtenção da matriz de rigidez global, esta função é aplicada, permitindo que só seja efetuada uma vez a mesma operação para todos os elementos.

De seguida é apresentada a função *walk elements* que executa um ciclo em todos os tipos de elemento e, para cada tipo de elemento, executa a função *walk array*.

```
def walk_elements( data, fn, args = 0 ):

  retval = 0;
  for el in list( data[ "element_a" ].keys() ):
    path = "element_a." + el;
    retval = retval + walk_array( data, path, fn, args );

  return retval;
```

Listing 3.14: Função *walk elements*

3.2.4.5 *Modules*

O ficheiro modules contempla a importação de módulos que não são, nada mais nada menos, que os ficheiros pré-definidos com toda a informação específica de cada elemento. Tendo já a função que permite correr todos os elementos de qualquer tipo, surge a necessidade de ir buscar a informação relativa ao elemento em questão quando se estiver a aplicar o método sob o ciclo.

Para tal, é necessário criar uma função que guarde esta informação para que, quando for preciso, baste especificar o módulo em questão para aceder aos dados pretendidos.

```
  ## Loading element related data
  modules_h[ 'element_types_a' ] = {};
  element_types_modules = modules_h[ 'element_types_a' ];

  ## elemnt objects
  element_files_a = [ "properties", "validate", "shape_funcs", "
     stiffness", "loads", "stresses" ];

  for el_type in element_types_a:
    dir = basic.feupFEM_HOME +"/src/elements/" +el_type;
    element_types_modules[ el_type ] = {};
    for i_file in element_files_a:
      try:
        element_types_modules[ el_type ][ i_file ] =
            SourceFileLoader( i_file, dir +"/" +i_file +".py" ).
            load_module();
      except Exception as e:
```

```
msg = 'Fail to load "' +el_type +"/" +i_file +'.py".';
print ( "ERROR: " +msg );
log.error( msg );
retval += 1;
```

Listing 3.15: Excerto da função de importação dos módulos de cada elemento

O ciclo apresentado anteriormente corre todos os tipos de elementos, guardados no diretório *elements* e, correndo cada ficheiro presente torna-se possível guardar cada um destes dentro de um objeto, chamado *modules_h*, que os armazenará de acordo com o seu tipo de elemento, para que posteriormente o seu *parsing* seja de fácil execução.

```
el_module = modules[ 'element_types_a' ][ el_type ]
properties = el_module[ "properties" ].data;
nnode = properties[ "nnode" ]
```

Listing 3.16: Exemplo de *parsing* da informação dos módulos de cada elemento

3.2.4.6 *Material*

De forma a que as características materiais de cada elemento possam ser usadas, é necessário estabelecer a ligação entre o cálculo propriamente dito e estas características que estão definidas no diretório *materials* como foi dito anteriormente. Dentro deste ficheiro *material*, é definida uma função que cria a ligação entre a identificação do material propriamente dito, inicialmente definida no ficheiro de *input*, com as propriedades do mesmo, para que seja possível, da mesma forma que foi realizado com os módulos de cada elemento, carregar o código necessario associado a cada elemento envolvido no domínio.

```
material_h = {};

def get( id ):

  material = 0;

  global material_h;
  if ( id in material_h ):
    material = material_h [ id ];
  else:
    file = basic.feupFEM_HOME +"/src/materials/" +id + "/
       properties.py";
    try:
```

```
        material_module = SourceFileLoader( id, file).load_module
            ();
        material = material_module.get();
        material_h [ id ] = material;
      except Exception as e:
        msg = 'Fail to load material "' +id +'".';
        print ( "ERROR: " +msg );
        log.error( msg );

  return material;
```

Listing 3.17: Função de ligação do material

3.2.4.7 Aplicação do método

Nesta fase do programa está tudo perfeitamente definido e preparado genericamente para que o método possa ser aplicado a qualquer elemento sem grandes alterações no programa todo quando novos elementos surgirem. Para efetuar todos os cálculos necessários recorreu-se ao livro do Método dos Elementos Finitos do professor Álvaro Azevedo [10], usado nas aulas de Ánalise Avançada de Estruturas.

Os ficheiros *stiffness, load_concentrated, load_edge_distributed, loads, contraints, solver e results* , presentes no diretório *kernel*, são a plena aplicação do método dos elementos finitos. De seguida é dado um pequeno exemplo de como se processa esta fase do programa até à sua demonstração de resultados.

A título de exemplo, será usado o tipo de elemento *EL_stress_4N_4E*, sendo este o elemento usado no programa.

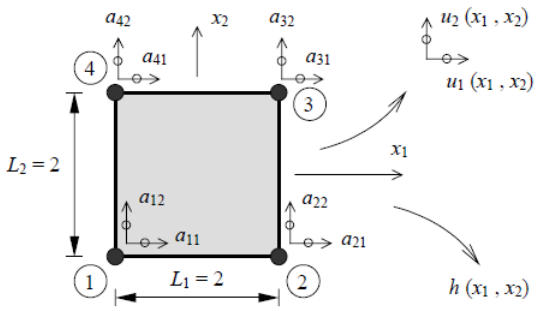

Figura 3.16: Exemplo de um elemento finito quadrado de 4 nós

Para uma discretização bem sucedida de um elemento, com base no conhecimento adquirido no estudo do o método dos elementos finitos, vários cálculos têm de ser efetuados para que toda a informação necessária à análise do mesmo esteja pronta a ser usada. Cada elemento tem as suas especificidades, no entanto, nesta fase, colmatadas todas as diferenças entre eles, todos podem ser tratados da mesma forma e calculados da mesma maneira.

Metodologia pragmática ao cálculo de um elemento:

1. Cálculo das funções de forma
2. Obtenção das matrizes de rigidez e deformação
3. Cálculo da matriz de rigidez do elemento no sistema de eixos geral
4. Espalhamento da matriz do elemento na matriz geral

Este procedimento de cálculo é muito resumido, mas é uma perspetiva simples de entender o encadeamento de resultados, visto que o objetivo desta dissertação não é estudar aprofundadamente o cálculo de um elemento finito, mas sim perceber que contribuições deste são necessárias para o desenvolvimento do resto do programa.

Aplicando os pontos anteriormente definidos, a função que calcula cada elemento pode ser definida dessa mesma maneira. De seguida pode-se observar as funções necessárias para construir a rigidez de cada elemento e completar o espalhamento da matriz de rigidez global. Dentro do ficheiro *stiffness*, função *build_each* é aplicada dentro do já explicado *walk array* que permite que o cálculo seja feito apenas uma vez, executando-o dentro de um ciclo um número de vezes correspondente ao número de elementos presentes. Todos os valores calculados neste processo que sejam necessários posteriormente, são guardados num objeto chamado *el_info*, específico do elemento, para que mais tarde seja possível aceder a esta informação.

```
def build_each( el, args ):
  global is_first_elem
  data = args[ 0 ];
  modules = args[ 1 ];
  res = args[ 2 ];
  el_type = args[ 3 ];
  el_type_info = args[ 4 ];
  el_type_info[ "nelem" ] += 1;

  el_module = modules[ 'element_types_a' ][ el_type ];

  ### el_info will contain information associated with each
     element
```

```
el_info = {};

build_shape_func_derivate_data( data, el_module, res, el,
    el_type, el_info );

build_elasticity_matrix( data, el_module, res, el, el_type,
    el_info );

build_BtDB( data, el_module, res, el, el_type, el_info );

build_el_stiffness( data, el_module, res, el, el_type, el_info
    , el_type_info );

push_to_K_matrix ( data, el_module, res, el, el_type, el_info
    );

el[ "stiffness_info" ] = el_info;
```

Listing 3.18: Função de cálculo da matriz de rigidez

Como foi já anteriormente referido na parte referente ao sub-diretório *extend*, para que a assemblagem de elementos finitos seja possível, cada nó da estrutura assemblada, no referencial global, tem de estar numerado para que a rigidez, força ou deslocamento associado a um dado grau de liberdade seja introduzido de uma maneira correta na respetiva matriz. Assim sendo, é necessário seguir uma convenção para que toda a gente que usufrua e contribua para o projeto tenha uma fácil interpretação. A convenção adotada é demonstrada na figura seguinte.

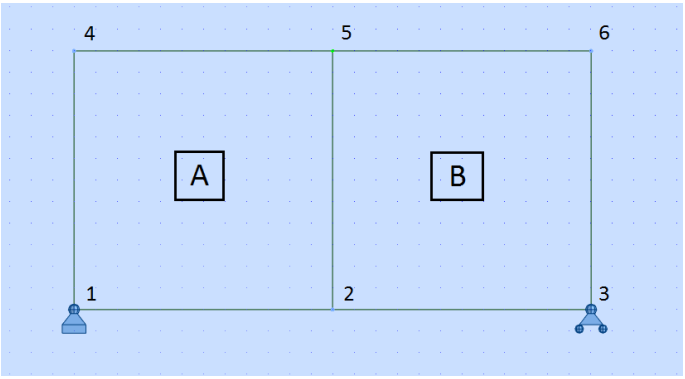

Figura 3.17: Exemplo da convenção adotada no cálculo de elementos finitos

Deste modo, procedido o cálculo das matrizes de rigidez de todos os elementos no referencial local, é necessário efetuar o espalhamento das a matrizes de rigidez de cada elemento pela

matriz global da estrutura. Computacionalmente, é possível inserir diretamente o valor de rigidez correspondente a um certo grau de liberdade no seu consequente índice na matriz de rigidez global, no entanto, para efeitos de compreensão, é explicado resumidamente como se procede à assemblagem tradicional dos mesmos. Tome-se como exemplo a estrutura apresentada na figura 3.17. Neste caso foram calculados 2 elementos de 4 nós. Consequentemente, cada um apresenta uma matriz de rigidez de 8 linhas e 8 colunas, correspondente à totalidade de graus de liberdade do mesmo. Quando transformadas num referencial global, a respetiva matriz transforma-se numa matriz de 12 por 12, visto que a estrutura contém 6 nós.

Matriz de rigidez no referencial local do elemento A:

$$K_A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} & A_{16} & A_{17} & A_{18} \\ A_{21} & A_{22} & A_{23} & A_{24} & A_{25} & A_{26} & A_{27} & A_{28} \\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & A_{36} & A_{37} & A_{38} \\ A_{41} & A_{42} & A_{43} & A_{44} & A_{45} & A_{46} & A_{47} & A_{48} \\ A_{51} & A_{52} & A_{53} & A_{54} & A_{55} & A_{56} & A_{57} & A_{58} \\ A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} & A_{67} & A_{68} \\ A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} & A_{78} \\ A_{81} & A_{82} & A_{83} & A_{84} & A_{85} & A_{86} & A_{87} & A_{88} \end{bmatrix} \tag{3.1}$$

Atendendo à numeração anteriormente definida, tomando como exemplo novamente o elemento A, a matriz de rigidez no referencial global será agora demonstrada. A este elemento estão associadas as rigidezes dos nós 1,2,5 e 4 na direção horizontal e vertical, portanto, é nesses índices da matriz que os valores se inserem.

Matriz de rigidez no referencial geral do elemento A:

$$K_A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & 0 & 0 & A_{15} & A_{16} & A_{17} & A_{18} & 0 & 0 \\ A_{21} & A_{22} & A_{23} & A_{24} & 0 & 0 & A_{25} & A_{26} & A_{27} & A_{28} & 0 & 0 \\ A_{31} & A_{32} & A_{33} & A_{34} & 0 & 0 & A_{35} & A_{36} & A_{37} & A_{38} & 0 & 0 \\ A_{41} & A_{42} & A_{43} & A_{44} & 0 & 0 & A_{45} & A_{46} & A_{47} & A_{48} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A_{51} & A_{52} & A_{53} & A_{54} & 0 & 0 & A_{55} & A_{56} & A_{57} & A_{58} & 0 & 0 \\ A_{61} & A_{62} & A_{63} & A_{64} & 0 & 0 & A_{65} & A_{66} & A_{67} & A_{68} & 0 & 0 \\ A_{71} & A_{72} & A_{73} & A_{74} & 0 & 0 & A_{75} & A_{76} & A_{77} & A_{78} & 0 & 0 \\ A_{81} & A_{82} & A_{83} & A_{84} & 0 & 0 & A_{85} & A_{86} & A_{87} & A_{88} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.2}$$

Procedendo da mesma maneira para a obtenção da matriz de rigidez do elemento B, a matriz de rigidez global da estrutura será composta pela soma das matrizes dos dois elementos no referencial global.

Vetor das forças nodais equivalentes do elemento A no referencial local:

$$F_A = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \\ F_5 \\ F_6 \\ F_7 \\ F_8 \end{bmatrix} \tag{3.3}$$

Vetor das forças nodais equivalentes do elemento A no referencial global:

$$F_A = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \\ 0 \\ 0 \\ F_5 \\ F_6 \\ F_7 \\ F_8 \\ 0 \\ 0 \end{bmatrix} \tag{3.4}$$

O vetor das forças nodais equivalentes global da estrutura será determinado pela soma dos vetores dos dois elementos no referencial global.

Com este pequeno exemplo é esperado que se perceba que, conhecendo apenas a posição de um dado elemento na estrutura, a sua inserção na matriz global será feita de um modo fácil e natural.

3.2.4.8 Solver

O ficheiro solver é simplesmente o local onde se vai localizar o cálculo necessário à resolução completa da estrutura. Quando se pretende analisar um pórtico, uma viga ou um pilar, o que se espera obter são os deslocamentos e tensões a estes associados, nos pontos desejados. Para tal, procede-se a resolução de um sistema de equações matricial. Este sistema, já bem conhecido, consiste no produto matricial da rigidez da estrutura pelos deslocamentos a ela associados que por sua vez será igual ao vetor das forças nodais equivalentes.

Importando então os vetores e matrizes já determinadas, necessárias para este cálculo, a determinação dos deslocamentos será feita através da resolução de um sistema de equações matricial normal do tipo Ax=b.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \quad (Ax=b)$$

Figura 3.18: Sistema de equações matricial genérico

O vetor de deslocamentos será o vetor que contém as incógnitas do sistema, no entanto, alguns destes valores podem ser previamente conhecidos. Se a um dado grau de liberdade estiver associado um apoio ou um assentamento, o valor do deslocamento tem de ser inserido no vetor, no grau de liberdade correspondente. No caso de um apoio, o valor a introduzir será zero pois o deslocamento nessa direção está impedido. No caso de um assentamento, basta inserir esse mesmo valor. Procedendo a estas operações, obtém-se facilmente o vetor de deslocamentos de toda a estrutura.

Este método de resolução do sistema de equações é um método simples, mas não é de todo o mais eficiente. Em casos reais, o número de nós necessários ao cálculo de uma estrutura pode chegar à casa dos milhares, traduzindo-se numa quantidade incomensurável de operações de cálculo em que a quantidade de memória consumida é astronómica. Uma das particularidades da distribuição dos valores da matriz de rigidez é que esta se torna simétrica quando definida. Ao explorar esta característica é possível economizar recursos informáticos, evitando-se assim o cálculo, o armazenamento dos termos do seu triângulo inferior e todas as operações que nestes seriam efetuadas. Para melhorar o cenário, é preciso atentar que grande parte dos elementos da matriz assumem o valor de zero, aperfeiçoando ainda mais o cálculo.

Um dos métodos mais usados na resolução de sistemas de equações na análise de elementos finitos é o chamado de *Frontal Solver*. Esta técnica é uma variação da eliminação de *Gauss* e é usada na resolução de sistemas de equações lineares de matrizes esparsas.

Este método de resolução de sistemas é mais um de muitos exemplos de entradas que podem vir a ser encaixadas neste modelo. Necessitando apenas da informação referente à rigidez, forças e deslocamentos, tal como um sistema de equações básico, este também se encaixa na perfeição neste projeto.

Para a inclusão deste tipo de resolução de sistemas, basta incluir o mesmo neste diretório kernel e depois, quando for necessário aplicar o *solving*, a função que define este novo sistema de equações deve ser chamada no ficheiro *fem.py* que controla todas as funções necessárias ao cálculo do programa.

3.2.4.9 *Results*

Quando resolvido o sistema de equações $K \times a$ = F num referencial global, isto é, todas as matrizes e vetores associados à totalidade de graus de liberdade da estrutura, é possível determinar, em qualquer ponto do elemento, o seu estado de tensão e deformação. Este passo, finaliza todo o processo de cálculo envolvido no programa. Conhecidos já os deslocamentos associados a cada elemento, conhecida a matriz de elasticidade em causa, e calculada a respetiva matriz de deformação nos pontos desejados, a resolução da estrutura está completa.

Este processo de cálculo, dentro do ficheiro *results*, tal como na determinação da matriz de rigidez, é também efetuado elemento a elemento, portanto, recorre-se de novo ao *walk array* para determinar todas as grandezas necessárias para uma conveniente análise de resultados. Depois de calculadas todas as grandezas, estas são empurradas para uma matriz de resultados para que posteriormente esta possa ser exportada para o ficheiro *JSON* de saída.

3.2.5 *Main file*

O *main file*, neste caso chamado de *fem.py*, é o ficheiro principal do programa e aquele que assume o controlo sobre tudo que se passa no mesmo, mais especificamente no que é executado. Este ficheiro é o esqueleto do programa e contém especificamente as funções necessárias, na ordem correta, para que o programa corra com sucesso. Com todos os cálculos efetuados anteriormente, em diretórios distintos, este local resume-se à chamada destas mesmas funções que perfazem o cálculo requerido. De seguida será feita uma análise a diferentes partes que compõem este ficheiro que é necessário conhecer para um bom entendimento de como corre o programa.

Todos os módulos necessários para se proceder à execução de funções e obtenção de ficheiros têm de ser importados inicialmente.

```
import os;
import sys;
import json;
import numpy as np
from kernel import log, modules, basic, extend, show, validate,
    kernel, stiffness, loads, constraints, solver, results;
from pprint import pprint;
```

Listing 3.19: Módulos necessários

Para executar o programa, é necessário abrir a janela de comandos e escrever os 5 argumentos descritos na função seguinte, correspodentes ao nome do *main file*, ao *input*, que neste caso será sempre um ficheiro *JSON* com o nome de *data.json*, e ao *output*, que pode ser um ficheiro de resultados *res.json* ou uma demonstração gráfica através da plataforma matlab, implicando que o último argumento assuma o nome de *matlab*. Esta função garante simplesmente que, em caso de engano do utilizador, é informada a maneira como deve ser executado o programa.

```
def f_usage( retcode ):
  print ( "Usage: fem.py -i data.json -o res.json" );
  f_end( retcode );
  return;
```

Listing 3.20: Execução do programa

Posteriormente, é altura de chamar as funções que constroem o esqueleto do programa. Para efeitos de exemplo, são demonstradas de seguida as funções necessárias quando se lida com os diferentes tipos de caso de carga. É necessário realizar um ciclo nestes e, para cada um deles calcular as respetivas forças nodais equivalentes, resolver o sistema de equações para determinar os deslocamentos da estrutura e por fim obter os resultados das grandezas obtidas.

```
### Cycle all loads and compute associated results
loads_n = len ( data[ "loads_a" ] );
for i_load in range( loads_n ):
  print ( "------------ i_load:" , i_load );

  ### Create Load Vector
  if ( loads.build( data, el_modules, res, i_load ) ):
    f_end( );

  ### Solve system of equations
```

```
  if ( solver.solve( data, el_modules, res ) ):
    f_end( );

  ### Compute results
  if ( results.build( data, el_modules, res ,i_load ) ):
    f_end( );

### Show results or call mapper to visualizer and call
    visualizer
if ( show.data( data, el_modules, res, input_file, output ) ):
  f_end ( );
```

Listing 3.21: Execução do programa

Como é possível observar, a manipulação das funções nesta fase é de extrema facilidade pelo que, quando surgir a necessidade de chamar outras ou substituir alguma presente, basta importar a mesma e chamá-la no sítio correto. Se eventualmente a função chamada não estiver definida, o programa é terminado recorrendo à função *f_end*.

Terminado o cálculo de todas as grandezas referentes aos diferentes casos de carga, a última função do programa corresponde a função *show.data* que permite a criação do ficheiro de resultados *JSON* ou a exibição dos resultados no *matlab*, dependendo do argumento usado na execução do programa.

3.3 *Output*

Com um significado oposto ao já definido conceito de *Input*, o *Output*, termo que também foi importado da língua inglesa e que é vastamente usado no ramo da tecnologia, traduzido, assume um significado de saída de algo, neste caso, dados.

Por definição, qualquer tipo de informação que seja processada e gerada por um computador ou dispositivo semelhante pode ser considerada como *Output*. Isto inclui informação gerada ao nível de um software, como é o caso, quando se obtém resultados de cálculos efetuados, ou até mesmo num nível mais físico, quando se imprime um documento por exemplo. Um exemplo básico da saída de dados de um software é o de um programa de uma calculadora que produz o resultado de uma operação matemática. Um caso mais complexo pode ser demonstrado com uso de um motor de busca como a *Google* que compara palavras chave com milhões de páginas numa página *web*.

Qualquer dispositivo que produza uma saída de dados física do computador é correntemente chamado de dispositivo de saída. Nos dias que correm, a interação com estes faz parte do quotidiano de cada pessoa. O dispositivo de saída mais conhecido e usado é o monitor de computador

que tem a capacidade de exibir os dados desejados num ecrã. Além deste, pode-se tomar também como exemplos a impressora de um computador ou colunas de som que têm a capacidade de transmitir a informação pretendida até ao utilizador.

Do mesmo modo que se usou um ficheiro *JSON* como interface de entrada, recorre-se também ao mesmo para perfazer a interface de saída. Repare-se que todo o cálculo do programa se encontra delimitado, tanto na sua entrada como na saída, por dois ficheiros desta qualidade que, por definição, têm uma génese neutra. Um projeto estruturado deste modo permite o uso de várias plataformas e interfaces gráficas, associadas a qualquer tipo de linguagem, para a demonstração de resultados. No trabalho desenvolvido nesta dissertação, os resultados, depois de armazenados num *JSON*, podem ser visualizados através de um programa que consiga fazer o *plotting* das coordenadas dos vários pontos da estrutura com os valores das grandezas pretendidas a estes associadas. Interpolando os valores sobre domínio da estrutura, é possível obter nuvens de valores para uma análise mais tangível. Optou-se por usar um software conhecido e muito avançado chamado de *Matlab* que permite executar os comandos anteriormente referidos e dispõe de uma boa interface para a apresentação dos resultados. É esperado que, com o passar dos anos, da mesma maneira que as informações de entrada do programa podem melhoradas com a criação de interfaces gráficas por exemplo, as informações de saída tenham o mesmo fim.

3.3.1 Conteúdo do ficheiro *JSON* de saída

Findados os cálculos de todos os elementos, chega a fase destes serem exportados para um ficheiro *JSON*, independentemente de serem utilizados para uma demonstração gráfica ou não. Para que os dados possam ser expostos de uma maneira correta, estes têm de estar também organizados de um modo eficaz para que não exista qualquer tipo de dificuldade no seu acesso. Antes de organizar a informação presente, é pertinente definir o que deve constar no ficheiro para depois sim ter uma estruturação acertada. Os valores obtidos ao longo do programa que se destinam à visualização estão neste momento armazenados numa matriz de resultados, devidamente organizados por grandezas e casos de carga. No presente caso, lidando com um problema de estado plano de tensão, foram obtidos no total 5 valores de grandezas para cada ponto estudado. Estas grandezas correspondem aos deslocamentos de cada ponto na direção x (Δ_x) e na direção y (Δ_y), às tensões normais na direção x (σ_x) e na direção y (σ_y) e às tensões tangenciais no plano xy ($\tau_{x,y}$). Os pontos selecionados para o cálculo destas grandezas correspondem aos nós e aos pontos de gauss de cada elemento.

A função de seguida apresentada foi definida no ficheiro que armazena as funções de demonstração de resultados, chamado de ficheiro *show*, dentro do diretório *kernel*.

```
def dump_to_json( res , out_file ):
  ## Fetch data from res and create out_file
```

```
  results_matrix = res["results_matrix"].tolist()

  with open(out_file , "w") as fp:
    json.dump( { "results": results_matrix}, fp,
                indent=2, separators=(',', ': ') )


  return;
```

Listing 3.22: Função *dump to JSON*

Conhecendo perfeitamente as dimensões e conteúdo da matriz de resultados, o seu parsing torna-se mais uma vez bastante acessível. Cada coluna desta matriz corresponderá a uma grandeza específica, pelo que, aquando da necessidade de a estudar, basta selecionar o índice desta mesma coluna para que seja possível obter todos os valores calculados desta grandeza em todos os pontos estudados. Cada linha desta matriz está associada a um ponto de coordenadas conhecidas, seja um nó ou ponto de gauss, que servirá posteriormente como argumento numa função de demonstração gráfica de resultados. Estas coordenadas, no caso bidimensional, encontram-se presentes nas primeiras 2 colunas, seguidas das 5 grandezas que lhes são inerentes vezes o número de casos de carga presentes. No caso seguinte, usufruindo de 2 casos de carga, a matriz de resultados apresentará 12 colunas.

```
"results": [
    [
      0.0,
      0.0,
      0.0,
      0.0,
      15.047014091430349,
      2.7167942042927913,
      3.196862239415424,
      0.0,
      0.0,
      27.912131713227147,
      13.241962419590283,
      13.710509348807204
    ],
```

Listing 3.23: Exemplo da primeira linha da matriz de resultados no ficheiro *JSON*

3.3.2 Transversalidade do *Output*

Uma vez mais, e agora relativamente à saída de dados, o comportamento da mesma vai ser idêntico ao da entrada. Estabelecendo a ligação dos cálculos com o exterior através de um *JSON* é permitido um mapeamento descomplicado dos valores pretendidos para qualquer ficheiro. A neutralidade inerente a este formato possibilita um encaixe natural de diferentes interfaces que, por sua vez, proporcionam uma análise mais percetível dos valores em estudo. Esta fase é uma das mais importantes do programa pois é quando efetivamente se torna exequível a observação, por exemplo, de tensões ou deslocamentos, sobre o domínio da estrutura. Esta possibilidade torna a análise da estrutura mais simples e direta, contribuindo também para a sensibilidade de quem a executa. Nos dias que correm, existem inúmeros *open source software* que permitem realizar este tipo de análise. Desta forma, numa fase inicial, recomenda-se que se recorra a um destes para a concretização desta análise pois estes são de fácil acesso. No entanto, na presente dissertação, os valores obtidos são demonstrados através do programa *MATLAB*

3.3.2.1 *MATLAB / Octave*

Do grande lote de programas disponíveis para a concretização dos resultados, o *software* escolhido foi o *MATLAB*. Uma boa alternativa a este seria o *GNU Octave*, um *open source software* que suporta a linguagem *MATLAB* e que é na verdade, 95% idêntico a este. Assim sendo, optou-se por usar esta mesma plataforma *open source* por não implicar qualquer custo e conseguir executar as mesmas tarefas.

Passa-se agora a explicar um pouco do que é *software* usado e de como este se encaixa e funciona no programa.

O *MATLAB*, abreviatura de *MATrix LABoratory*, é um software interativo de alta performance indicado para o cálculo numérico. Além de se tratar de um ambiente de computação numérica multi-paradigma, é também uma linguagem de programação de quarta geração que permite a manipulação de matrizes, plotagem de funções e dados, implementação de algoritmos, criação de interfaces de usuário e interface com programas escritos em outras linguagens como por exemplo C, C++, *Java, Python* e *Fortran*.

As principais vantagens de recorrer a este *software* são:

- Linguagem de alto nível
- Possui um grande número de bibliotecas de funções pré-definidas
- Fácil compreensão e uso das funcionalidades gráficas para visualização de dados
- Largamente divulgado e usufruído em faculdades, como é o caso da FEUP

3.3.2.2 *Show*

Possuindo toda a informação necessária armazenada no ficheiro *JSON*, chega a altura desta ser utilizada e demonstrada. Para tal, dentro do ficheiro show onde se criou a função *dump_to_json*,

criou-se também uma função que trata de organizar a informação já existente e a mapear para o formato de um ficheiro que possa ser lido pela plataforma *Octave*, ou seja, um ficheiro do tipo *.m* Assim sendo, depois de aberto o ficheiro de resultados *JSON*, a informação é escrita num ficheiro *results_data.m* que terá de possuir a organização necessária a fim de conseguir ser interpretado por outro ficheiro, chamado *show.m*, que conterá toda a inteligência necessária para executar a função de *plotting* às grandezas pretendidas. Ou seja, no final obtém-se 2 ficheiros matlab, um com informação de dados perfeitamente estruturada, e outro com a capacidade de os processar e demonstrar.

```
show = -1;
while ( show ~= 0 )

  clc
  disp (menu_1);
  disp ('  ')
  show = input ( 'Which combination? ' );
  if ( show == 0 )
    exit ( 0 );
  end
  option = 1;
  while ( option ~= 0 )
    clc
    disp ( ['Combination name:' ,  menu_1( show + 1, 5:end ) ] )
    disp ('  ')
    disp (menu_2);
    disp ('  ')
    option = input( 'Which option?')

    if (option == 0)
      break
    else

      cindex = 0
      comb_i = combinations(show , :)
      comb_i_len = length(comb_i)
      for i = 1:comb_i_len
        cindex += comb_i(i)*load(:,:,i)(:,option)
      end
```

```
        p= patch( 'Faces', faces, 'Vertices', verts, '
            FaceVertexCData', cindex,'FaceColor', 'interp' )

        colorbar
        colormap(jet)
    end
  end
end
```

Listing 3.24: Ficheiro *show.m*

A função usada para a demonstração de resultados é a função *patch* que recebe três argumentos. O primeiro, chamado de *Faces*, é uma matriz com os pontos que definem o domínio do campo de resultados. Cada linha é formada pelos pontos que definem cada elemento, portanto, será observado um número de linhas igual ao número de elementos. O segundo, chamado de *verts*, é uma matriz composta pelas coordenadas dos pontos calculados ao longo do programa que contêm uma grandeza associada, seja ela uma tensão ou deslocamento. Por fim, o terceiro argumento, chamado de *cindex*, é um vetor coluna que possui os valores da grandeza pretendida associada a cada ponto presente na matriz *verts*. Portanto, estes dois últimos argumentos têm o mesmo número de linhas.

Para definir as cores que revelam a variação de resultados usa-se a função colormap. Para adicionar uma barra lateral que fará a correspondência entre o valor e uma cor, basta empregar a função colorbar.

Para uma boa interação com o utilizador nesta fase de análise de resultados, o *script* criado permite que seja escolhida nesta fase, a combinação desejada previamente definida e uma grandeza em questão. Assim sendo, foi criado um menu que aparecerá quando o programa for corrido como se pode observar nas imagens seguintes.

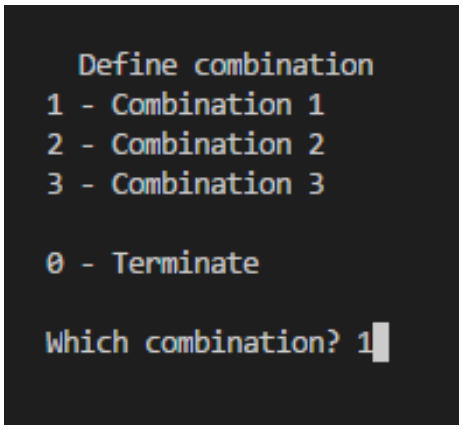

Figura 3.19: Menu 1

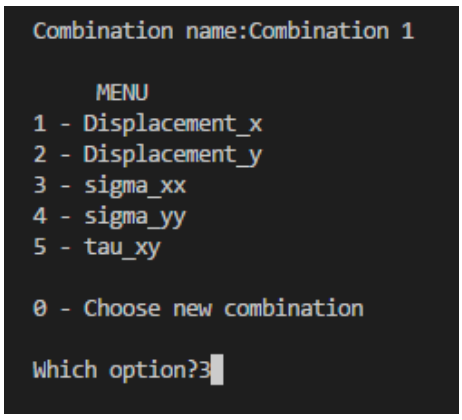

Figura 3.20: Menu 2

O resultado final do programa de elementos finitos desenvolvido neste trabalho assume a forma apresentada na figura 3.21. O exemplo referido é uma estrutura composta por 8 elementos quadriláteros e um total de 16 nós. Esta encontra-se submetida a um carregamento distribuído de 10 KN/m dos nós 13 ao 16 e possui um apoio duplo no nó 1 e um simples no nó 4.

A particularidade do uso desta função *patch* para esta análise é que esta permite definir concretamente o domínio da estrutura com que estamos a lidar. Desta forma, é possibilitada a demonstração dos resultados associados ao cálculo de estruturas com buracos, como é exemplificado na figura 3.21.

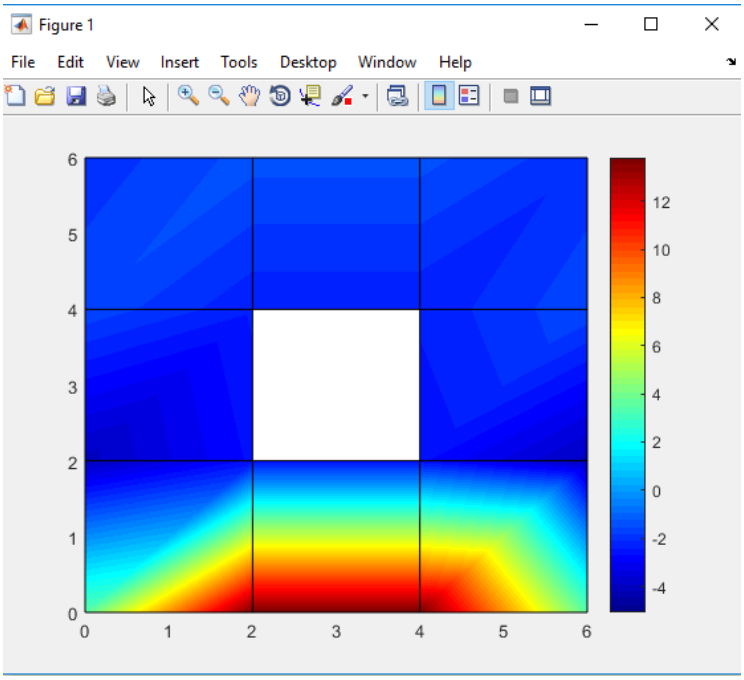

Figura 3.21: Exemplo da distribuição dos valores das tensões normais na direção x de uma dada estrutura

Para a validação dos valores obtidos, é comum fazer-se testes ao programa para que se obtenha a confirmação que este funciona como é devido. Dado o ficheiro *JSON* de entrada com uma informação específica, o programa consegue correr e no final gerar outra vez um *JSON* com os valores referentes aos resultados obtidos. Usando os mesmos dados de entrada, recorrendo desta vez a um *software* de elementos finitos já existente, retira-se os valores obtidos à mesma estrutura, submetida às mesmas condições. Por fim, fazendo a comparação entre o ficheiro de resultados do programa criado e os resultados obtidos num *software* já existente o programa em questão pode ou não ser validado.

3.3.2.3 Programas de visualização de resultados de elementos finitos

Uma outra alternativa ao uso de programas como o *MATLAB*, que requerem o conhecimento e escrita de código para que seja possível o *plotting* de funções e dados e a sua consequente observação, é a utilização de programas já existentes, criados especificamente para a demonstração de resultados de uma análise de elementos finitos.

Este tipo de programas é conhecido como *software* de pós processamento. Como o nome indica, após o cálculo de todos os elementos, obtidos os resultados, estes são processados para uma adequada visualização.

Uma rápida pesquisa na *Internet* permite perceber que existe um grande lote de oferta de programas deste género, desde *open source* a *software* pago. Neste instante, falta só perceber como se pode usufruir destes mesmos, nas condições presentes do programa desenvolvido nesta dissertação. Qualquer *software* desta categoria permite a importação direta de resultados e, entre estes, a única variação que pode ocorrer é no *upload* destes mesmos valores. A forma como estes têm de estar organizados e o formato em que têm de se encontrar são fatores que podem alterar com o *software* escolhido para importar resultados. Na maior parte dos casos, a organização exigida dos valores calculados é a que foi já admitida no *JSON* de saída. Se por acaso surgir a necessidade de dispor os resultados num formato diferente, esta situação não será um problema pois, como referido já referido neste capítulo, a conversão de um ficheiro de dados *JSON* para outro formato é de fácil exequibilidade, seja ele um formato de texto (TXT) ou *comma-separated values* (CSV), por exemplo.

Um exemplo de um possível *open source software* a usar para a visualização de resultados é o *Gmsh*. Este *software* livre é um gerador de malha de elementos finitos lançado pela *GNU General Public License*, com a sua última versão referente ao ano de 2017. O facto da sua utilização não incluir qualquer custo torna-o uma das ferramentas mais procuradas por engenheiros para a análise estrutural e não só.

O *Gmsh* possui a seu dispor 4 módulos:

- descrição geométrica
- geração de malha
- resolução
- pós-processamento

Uma das características que torna este programa tão procurado é a sua capacidade de permitir entrada de informações em cada um desses módulos. Esta pode ser feita através da interface gráfica de forma interativa, através de um arquivo ASCII ou utilizando uma API em C, C++ ou *Python*.

3.3.2.4 Possíveis contribuições

Nesta fase do trabalho é possível perceber que a existência de inúmeras ferramentas capazes de fazer uma simples geração de malha e um pós processamento de resultados simplificam a tarefa final do programa. No entanto, tendo em mente que o objetivo do mesmo é um crescimento natural ao longo dos anos, é esperado que, futuramente, possa surgir uma contribuição que perfaça estas tarefas sem que seja necessário recorrer-se a programas terceiros. Com os valores que o programa já fornece, é esperado que a entrada desta contribuição encaixe na perfeição e que permita uma grande interação com a geometria da estrutura em causa. A aplicação de rotações, zoom e translações são funções básicas neste tipo de análise, seja ela em 2 ou 3 dimensões. Com a capacidade de demonstrar a variação de tensões através de gráficos e distribuições de cores ao longo do domínio da estrutura, uma característica deste tipo de análise é a inclusão de animações que demonstram o comportamento da estrutura em causa quando submetida a certas condições. Estes são apenas alguns exemplos de possíveis contribuições a este projeto que são imprescindíveis a qualquer grande *software* de elementos finitos. Esta contribuição, obtida é facilmente encaixada no ficheiro show, onde já existe até o momento a função *dump_to_json* e *matlab* que são as opções disponíveis até ao momento. Posteriormente, no ficheiro *fem.py*, aquando da chamada do função data do ficheiro *show.py*, esta já conterá mais uma opção de demonstração de resultados, que poderá ser escrita como último argumento no comando de execução do programa, tornando o encaixe da mesma muito simples.

```
def data( data, modules, res, input_file, output ):
  retval = 0;

  if ( -1 != output.find ( '.json' ) ):
    dump_to_json( res, output );
    print( 'Created file "' +output +'".' );

  elif ( output == 'matlab' ):
    out_file = "res.json" ;
    dump_to_json( res, out_file );
    retval = matlab(data, res, out_file  );
  else:
    print( 'ERROR: currently we can only create a json results
       file or call matlab.' )
    retval = 1;

  return retval;
```

Listing 3.25: Função data de apresentação de resultados

Capítulo 4

Regras de submissão de um novo elemento

O futuro e sucesso de um projeto *open source* está diretamente ligado ao número e à qualidade de contribuições que este recebe. Projetos de grande dimensão lidam com mudanças todos os dias e é imperativo que exista um grande controlo e uma boa organização por parte dos autores para que nada prejudique o projeto em causa. Quando se lida com contribuições de pessoas externas ao projeto, desconhecidas ou não, as novas entradas que estas providenciam têm de ser testadas de modo a que sejam aceites e integradas no programa. Para este efeito, são realizados testes de aceitação de entrada, podendo desta forma averiguar-se se a contribuição em causa pode ou não ser encaixada. Neste capítulo procura-se explicar como funcionam os testes de aceitação e como deve proceder o contribuidor para que o novo elemento em questão seja aceite.

4.1 Testes de aceitação

Um teste de aceitação é uma fase do processo de testes em que é realizado um teste de caixa preta no programa antes da sua disponibilização ao mundo. O objetivo da realização dos testes é de verificar se o programa está funcional em relação aos seus requisitos originais.

Um teste de caixa preta é um teste de *software* para avaliar a saída de dados usando entradas de vários tipos. Evidentemente, quanto mais entradas forem fornecidas, mais completo será o teste. Idealmente, todas as entradas possíveis devem ser testadas, mas na grande maioria dos casos isso torna-se impossível devido à ampla quantidade de informação. Uma técnica normalmente usada para a realização deste tipo de testes é a escolha de um subconjunto de entradas o mais diversificadas possível para que grande parte dos casos sejam abrangidos. No caso de um programa de elementos finitos, uma boa maneira de se explorar o funcionamento do mesmo é criar diferentes casos de carga e geometrias variadas, com condições de apoio alteradas também. Quando se

prepara o programa para analisar uma estrutura submetida a diferentes cargas concentradas, diferentes cargas distribuídas, assentamentos e apoios diversificados, em diferentes cenários, e este apresenta resultados corretos, à partida será possível afirmar que este está funcional para qualquer cenário [9].

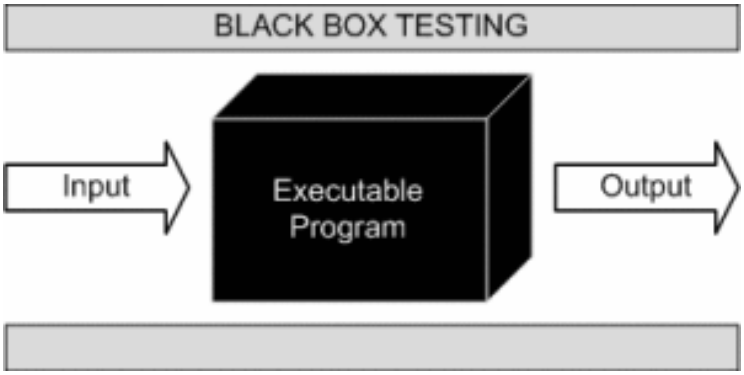

Figura 4.1: Teste de caixa preta

Este método de teste de *software* vai ser o utilizado para testar o programa e validar a entrada de um novo elemento, portanto, apresenta-se de seguida uma sequência de passos chave para que a sua exequibilidade seja mais simples.

1. Analisar todas as especificações e requisitos do programa. A maneira como os dados devem estar estruturados para a entrada é algo que tem de ser estudado.
2. Escolher dados de vários *inputs* válidos e também não válidos para ter a certeza que o programa os deteta.
3. Com os dados de vários *inputs*, recorrer a um *software* existente e obter valores de *output* corretos.
4. Criar casos teste com os *inputs* usados.
5. Executar os testes.
6. Comprarar os resultados obtidos com os valores admitidos como corretos.
7. Se forem detetados erros, corrigir e voltar a testar.

O procedimento anteriormente descrito é um método muito geral de fazer testes a um *software*. Quando se trabalha num ambiente aberto, este procedimento tem de ser repetido para cada nova entrada de forma a garantir que o programa está sempre funcional.

4.2 Requisitos para uma contribuição

A submissão de novas entradas de elementos para este projeto é a chave de todo o trabalho. Para tal, a maneira como uma pessoa deve contribuir deve ser explicada de forma a tornar este processo mais simples. Esta dissertação servirá então como guia para qualquer contribuidor, bastando seguir os passos posteriormente explicados.

4.2.0.1 Submissão do elemento

Quando se fala na submissão de um elemento, o que se espera obter para encaixar no programa são os vários ficheiros de código que perfazem a discretização do mesmo. Como foi já explicado no capítulo anterior, a entrada deste será natural, juntando-se ao diretório de outros elementos já calculados.

Assim sendo, o que se exige então a um contribuidor é que cumpra a sua submissão com os 3 passos seguintes:

- Submissão dos ficheiros de código referentes ao cálculo do novo elemento.
- Submissão de dados de *input* num ficheiro de texto.
- Submissão dos resultados do cálculo do novo elemento num ficheiro de texto recorrendo a um software já existente, de modo a que os valores sejam corretos, usando os dados de *input* para submissão.

4.2.0.2 Validação do elemento

Recebida toda a informação exigida na submissão de um novo elemento, chega agora a fase da validação do mesmo. Para tal, recorre-se aos testes de aceitação de entrada já anteriormente explicados. Se o teste não indicar qualquer erro, o elemento é aceite, caso contrário, o contribuidor é notificado do mesmo e aconselhado a alterar o código escrito. Os testes de aceitação a uma nova entrada seguirão a seguinte ordem de ações:

1. Converter os ficheiros de dados recebidos num formato que se encaixe no programa, neste contexto, num formato *JSON*
2. Incluir a informação do novo elemento no programa
3. Correr o programa e obter resultados
4. Comparar os resultados obtidos com o ficheiro de dados recebido respeitante aos resultados reais e confirmá-los também recorrendo a um *software*.
5. Se não houver diferença entre os valores de resultados calculados e os valores de resultados recebidos, o elemento está apto para ser incluído no projeto

4.3 Submissão de um elemento

Quando uma nova entrada passa em todos os testes de validação, chega a hora desta ser incluída no projeto. No âmbito de *open source*, todo o trabalho em si é armazenado num repositório numa plataforma online, como por exemplo no *Bitbucket* ou no *GitHub*, para que seja possível a interação de vários intervenientes no projeto.

Para uma melhor compreensão de como funciona e como é que se deve proceder para incluir novas entradas no projeto, são apresentados de seguida as várias fases para que as mesmas sejam incorporadas com sucesso.

4.3.0.1 Copiar repositório *Git*

O presente projeto, intitulado de *feupFEM*, foi armazenado na plataforma *Bitbucket* e é nesta que se baseiam os seguintes passos. O presente repositório pode ser acedido através do seguinte url `https://bitbucket.org/joaopiresmacedo/feupfem/src/master/`.

Existindo uma plataforma pronta para receber contribuições, é necessário estabelecer uma ligação à mesma a partir no sistema local, neste caso o computador em que se esteja a trabalhar. Assim sendo, o primeiro passo a tomar é copiar o repositório existente no *Bitbucket* para o sistema pessoal, habitualmente chamado de *cloning*. Quando um repositório é clonado, é criada automáticamente uma ligação entre o servidor do *Bitbucket* e o computador.

A clonagem do repositório é bastante simples de se executar. Acedendo ao url anteriormente disponibilizado, no canto superior do lado direito aparecerá um ícone onde se encontra escrito *clone*, que deverá ser premido. De seguida, aparecerá uma janela contendo um *link* como é apresentado na figura seguinte.

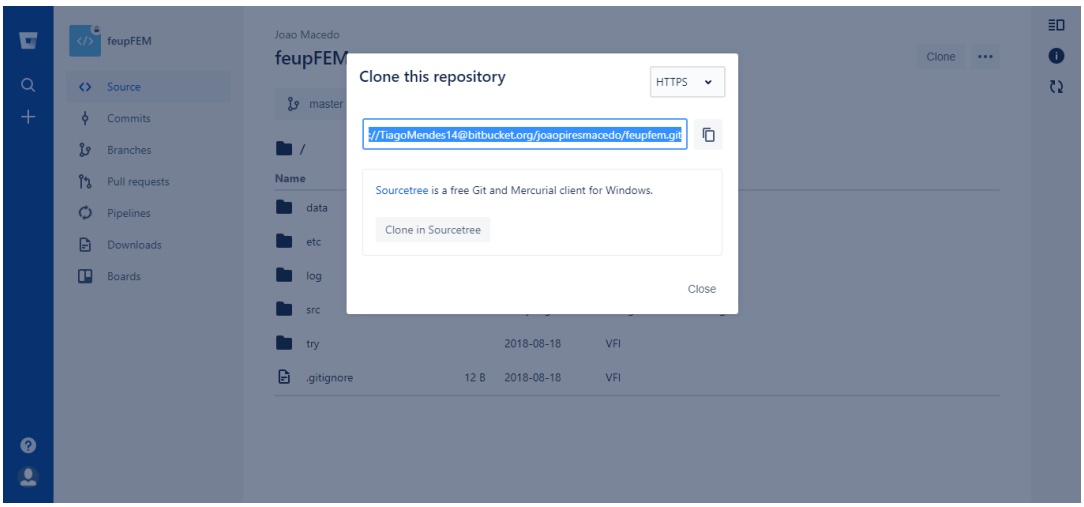

Figura 4.2: Clonagem do repositório

Criado o diretório onde se irá trabalhar, deve-se copiar o *link* apresentado, inserir o mesmo na janela de comandos e executar o comando. Executando todos os passos descritos o repositório encontra-se clonado e a ligação estabelecida.

```
git clone https://TiagoMendes14@bitbucket.org/joaopiresmacedo/feupfem.
    git
```

Listing 4.1: Exemplo do link de clonagem

4.3.0.2 Efetuar e adicionar mudanças

Num projeto que envolve várias pessoas, como é o caso do *open source*, é do interesse de todos que as várias alterações que a este são efetuadas sejam atualizadas com a maior frequência possível. Para tal, e tendo já estabelecida a ligação direta do computador ao servidor, é necessário fazer um *push* de todas as alterações que se quer incluir no projeto. Como diz o termo, as modificações serão empurradas do computador para o *Bitbucket*.

Para uma boa prática e simplicidade de compreensão de como atualizar o repositório, os comandos necessários a executar na janela de comandos, dentro do diretório em causa, são os seguintes:

- git status
- git add
- git commit
- git push
- git pull

Em projetos de grande escala, todos os dias são alterados e criados novos ficheiros e torna-se complicado saber todas as modificações que foram realizadas. Executando o comando git status, este apresenta todos os ficheiros alterados até à data, desde a última atualização do projeto.

Supondo que se criou um ficheiro novo, no diretório em questão, chamado de *results.py*, a execução do comando *git status* devolverá algo como o exemplo apresentado, dependendo do sistema operativo em causa.

```
git status
 On branch master
 Initial commit
 Untracked files:
 (use "git add <file>..." to include in what will be committed)
 show.py
```

```
nothing added to commit but untracked files present (use "git add" to
    track)
```

Listing 4.2: Exemplo git status

Após o conhecimento de todas as alterações, o passo seguinte é discriminar quais delas se pretende adicionar ao repositório. Assim sendo, executando o comando *git add*, seguido do ficheiro em causa, estas alterações serão transferidas para a chamada *Git staging area* que engloba todas as alterações que se pretende fazer o *commit*. Se a execução o comando for bem sucedido este não retornará nada. Se posteriormente se executar outra vez o comando git status, já não aparecerá nada pois todas as alterações que se queria atualizar foram adicionadas.

```
git add show.py
```

Listing 4.3: Exemplo git add

Com as alterações todas prontas a serem enviadas para o histórico do projeto, surge o comando *git commit*. Este comando é normalmente acompanhado com uma mensagem referente às alterações que foram realizadas. Quando este é executado, estas alterações são enviadas para o histórico de *commits* atualizando o projeto, mas ainda no sistema local, ou seja, no computador pessoal.

De forma a atualizar definitivamente o repositório no *Bitbucket*, abre-se novamente a janela de comandos e aplica-se o comando *git push*.

```
git push
 Counting objects: 3, done.
 Writing objects: 100% (3/3), 253 bytes | 0 bytes/s, done.
 Total 3 (delta 0), reused 0 (delta 0) To https://
    TiagoMendes14@bitbucket.org/joaopiresmacedo/feupfem.git
 * [new branch] master -> master
 Branch master set up to track remote branch master from origin.
```

Listing 4.4: Exemplo git push

Nesta fase o repositório do *Bitbucket* encontra-se atualizado e qualquer pessoa que esteja envolvida no projeto pode facilmente aceder às mais recentes modificações. Para que se possa trabalhar com estas alterações, o processo tem de ser invertido e desta vez é necessário transferir o projeto atualizado do *bitbucket* para o computador pessoal. Para tal, resta por fim executar o comando *git pull*.

```
git pull
 Fetching origin
 remote: Counting objects: 3, done.
 remote: Compressing objects: 100% (3/3), done.
 Unpacking objects: 100% (3/3), done.
 From https://TiagoMendes14@bitbucket.org/joaopiresmacedo/feupfem
```

```
1 file changed, 5 insertions(+)
```

Listing 4.5: Exemplo git pull

Capítulo 5

Exemplo da entrada de um elemento tipo no modelo

Tendo passado em todos os testes necessários, o elemento novo está pronto a ser incluído no modelo. Para a entrada deste novo tipo de elemento e, da mesma forma que foi efetuado para um elemento de 4 nós, criou-se o diretório *EL_stress_3N_3E* com uma dada estrutura, contendo os ficheiros *loads, properties, shape_funcs, stiffness, stresses*, referidos anteriormente no capítulo 3.3.2 Elements, mas adaptados agora a este tipo de elemento. Foi ainda criado um diretório de testes onde foi colocado :

- um ficheiro de dados usado como entrada para o programa de validacao usado, neste caso o *software* LISA 8.0.0.
- um ficheiro de resultados de referência, resultante do mapeamento do ficheiro de resultados que o programa de validação produziu.
- colocação de um ficheiro JSON em formato de entrada para o programa feupFEM
- colocação de um ficheiro *README* que contém a informação básica do processo de teste do elemento, como por exemplo, o tipo de elemento em questão, o *software* usado para confrontar os resultados obtidos e em que sistema operativo, e ainda o nome dos ficheiros usados no processo. O ficheiro de entrada no programa feupFEM terá a designação de *data.json* e o ficheiro de resultados de referência obtidos através *software* de validação será disposto num ficheiro chamado *refRes.json*

Toda a informação que seja específica deste elemento será incluída no sub-diretório correspondente a este elemento, dentro do diretório *elements*, o resto é tratado pelo programa. Como se trata de um elemento de uma geometria distinta dos elementos quadrangulares por exemplo, a aplicação da quadratura de Gauss no domínio de -1 a 1 já não pode ser aplicada, sendo necessária o cálculo da quadratura especifica a este elemento e a sua respetiva inclusão no programa, além das já incluídas funções de forma e informações referentes ao cálculo de ações e pontos de gauss, que

também elas são específicas deste elemento [11][15]. Neste ponto, o elemento pode ser incluído no projeto e calculado como anteriormente se procedia para um elemento quadrangular de 4 nós.

De forma a que o elemento incluído possa ser calculado, a informação de entrada referente ao mesmo têm de ser também incluída no ficheiro *JSON* de entrada. Esta informação, tal como qualquer outro elemento, têm de conter dados alusivos à sua geometria, material e todas as condições de carga e apoios em que este se encontra.

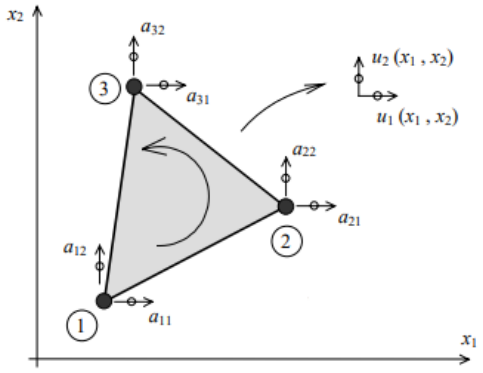

Figura 5.1: Exemplo de um elemento triangular de 3 nós

A juntar à informação já existente respeitante a outros elementos, surgirá agora o *input* necessário ao cálculo de um elemento triangular de 3 nós. Tal como já foi referido, estes dados estarão organizados de uma maneira específica para que o programa consiga correr com sucesso. A forma como todos os dados de entrada são obtidos é exatamente a mesma para qualquer elemento. Usufruindo da capacidade de obtenção de valores que um ficheiro *JSON* oferece, o parsing dos dados necessários será trivial.

```
{
    "coord_a":[
        [
            "P1",
            0,
            0
        ],
        [
            "P2",
            2,
            0
        ],
        [
            "P3",
            0,
            2
        ],
        [
            "P4",
            2,
            2
        ],
        [
            "P5",
            0,
            4
        ]
    ],
    "element_a":{
        "plane/EL_stress_3N_3E":[
            {
                "name":"E1",
                "material":"concrete/C25",
                "coord_a":[
```

```
                "P1",
                "P2",
                "P3"
            ],
            "height_a":[
                1
            ]
        },
        {
            "name":"E2",
            "material":"concrete/C25",
            "coord_a":[
                "P2",
                "P4",
                "P3"
            ],
            "height_a":[
                1
            ]
        },
        {
            "name":"E3",
            "material":"concrete/C25",
            "coord_a":[
                "P3",
                "P4",
                "P5"
            ],
            "height_a":[
                1
            ]
        }
    ]
},
"fixed_point_a":[
    {
        "point":"P1",
        "disp_a":[
            "F",
            "F"
```

```
            ]
        },
        {
            "point":"P2",
            "disp_a":[
                0,
                "F"
            ]
        }
    ],
    "loads_a":[
        {
            "name":"dead load",
            "concentrated":[
                {
                    "point":"P3",
                    "value_a":[
                        10,
                        0
                    ]
                }
            ],
            "edge_distributed":[
                {
                    "element":"E1",
                    "edge_a":[
                        "P1",
                        "P3"
                    ],
                    "value_a":[
                        [
                            0,
                            0
                        ],
                        [
                            0,
                            0
                        ]
                    ]
                }
```

```
            ]
        },
        {
            "name":"wind",
            "concentrated":[
                {
                    "point":"P3",
                    "value_a":[
                        0,
                        0
                    ]
                }
            ],
            "edge_distributed":[
                {
                    "element":"E1",
                    "edge_a":[
                        "P1",
                        "P3"
                    ],
                    "value_a":[
                        [
                            0,
                            -20
                        ],
                        [
                            0,
                            -20
                        ]
                    ]
                }
            ]
        }
    ],
    "combinations":[
        {
            "name":"Combination 1",
            "w":[
                1,
                0.5
```

```
            ]
        },
        {
            "name":"Combination 2",
            "w":[
                0.7,
                0.5
            ]
        },
        {
            "name":"Combination 3",
            "w":[
                1.2,
                0
            ]
        }
    ]
}
```

Listing 5.1: Exemplo de dados de entrada de um elemento triangular de 3 nós

Usando os dados apresentados anteriormente, calculou-se a estrutura e obteve-se os seguintes resultados referentes aos deslocamentos na direção horizontal.

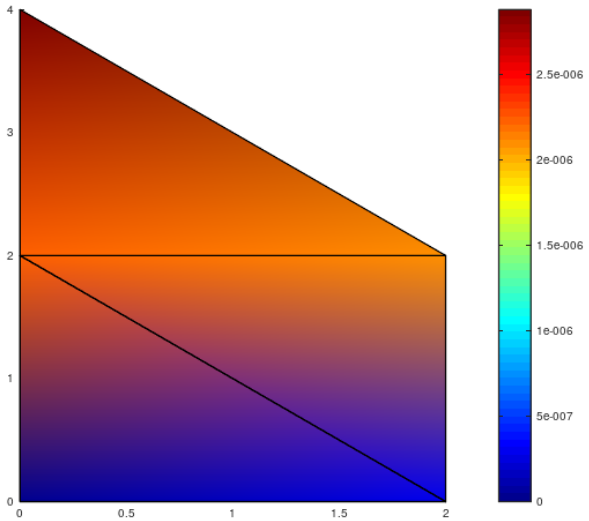

Figura 5.2: Deslocamentos na direção x do cálculo de elementos triangulares

Nesta fase estão reunidas todas as condições para que a mistura de vários elementos seja possível. Existindo a capacidade de calcular cada elemento individualmente, basta fazer a sua assemblagem e consequente cálculo de resultados. A forma como os diferentes tipos de elementos têm a capacidade de se misturarem, somando as contribuições de cada um no respetivo grau de liberdade é que torna toda a entrada de elementos neste programa natural.

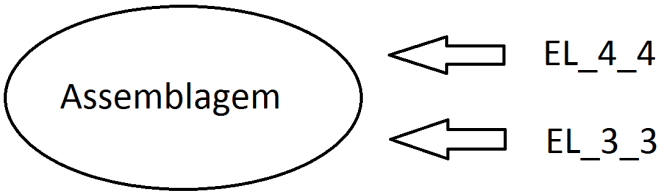

Figura 5.3: Processo de assemblagem

A título de exemplo, é apresentado se seguida um caso simples de uma assemblagem de um elemento quadrilátero de 4 nós e um elemento triangular de 3 nós.

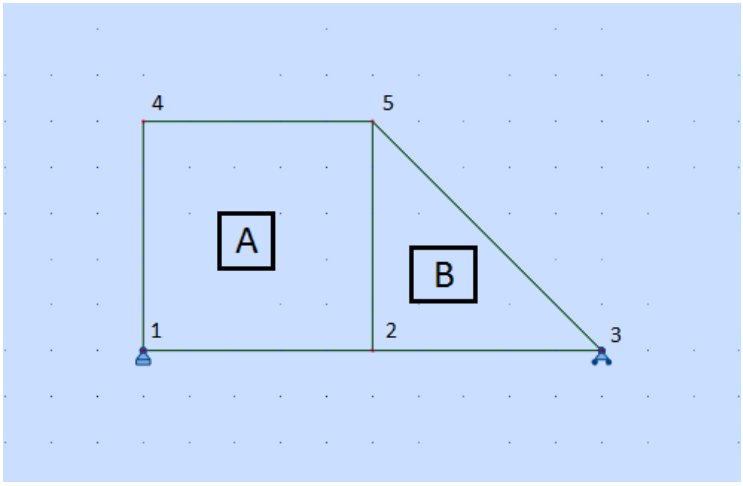

Figura 5.4: Estrutura composta por 2 elementos distintos

Matriz de rigidez do elemento A no referencial local:

$$K_A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} & A_{16} & A_{17} & A_{18} \\ A_{21} & A_{22} & A_{23} & A_{24} & A_{25} & A_{26} & A_{27} & A_{28} \\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & A_{36} & A_{37} & A_{38} \\ A_{41} & A_{42} & A_{43} & A_{44} & A_{45} & A_{46} & A_{47} & A_{48} \\ A_{51} & A_{52} & A_{53} & A_{54} & A_{55} & A_{56} & A_{57} & A_{58} \\ A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} & A_{67} & A_{68} \\ A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} & A_{78} \\ A_{81} & A_{82} & A_{83} & A_{84} & A_{85} & A_{86} & A_{87} & A_{88} \end{bmatrix} \tag{5.1}$$

Matriz de rigidez do elemento B no referencial local:

$$K_B = \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} & B_{15} & B_{16} \\ B_{21} & B_{22} & B_{23} & B_{24} & B_{25} & B_{26} \\ B_{31} & B_{32} & B_{33} & B_{34} & B_{35} & B_{36} \\ B_{41} & B_{42} & B_{43} & B_{44} & B_{45} & B_{46} \\ B_{51} & B_{52} & B_{53} & B_{54} & B_{55} & B_{56} \\ B_{61} & B_{62} & B_{63} & B_{64} & B_{65} & B_{66} \end{bmatrix} \tag{5.2}$$

Matriz de rigidez do elemento A no referencial global:

$$K_A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & 0 & 0 & A_{15} & A_{16} & A_{17} & A_{18} \\ A_{21} & A_{22} & A_{23} & A_{24} & 0 & 0 & A_{25} & A_{26} & A_{27} & A_{28} \\ A_{31} & A_{32} & A_{33} & A_{34} & 0 & 0 & A_{35} & A_{36} & A_{37} & A_{38} \\ A_{41} & A_{42} & A_{43} & A_{44} & 0 & 0 & A_{45} & A_{46} & A_{47} & A_{48} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A_{51} & A_{52} & A_{53} & A_{54} & 0 & 0 & A_{55} & A_{56} & A_{57} & A_{58} \\ A_{61} & A_{62} & A_{63} & A_{64} & 0 & 0 & A_{65} & A_{66} & A_{67} & A_{68} \\ A_{71} & A_{72} & A_{73} & A_{74} & 0 & 0 & A_{75} & A_{76} & A_{77} & A_{78} \\ A_{81} & A_{82} & A_{83} & A_{84} & 0 & 0 & A_{85} & A_{86} & A_{87} & A_{88} \end{bmatrix} \tag{5.3}$$

Matriz de rigidez do elemento B no referencial global:

$$K_B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & B_{11} & B_{12} & B_{13} & B_{14} & 0 & 0 & B_{15} & B_{16} \\ 0 & 0 & B_{21} & B_{22} & B_{23} & B_{24} & 0 & 0 & B_{25} & B_{26} \\ 0 & 0 & B_{31} & B_{32} & B_{33} & B_{34} & 0 & 0 & B_{35} & B_{36} \\ 0 & 0 & B_{41} & B_{42} & B_{43} & B_{44} & 0 & 0 & B_{45} & B_{46} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & B_{51} & B_{52} & B_{53} & B_{54} & 0 & 0 & B_{55} & B_{56} \\ 0 & 0 & B_{61} & B_{62} & B_{63} & B_{64} & 0 & 0 & B_{65} & B_{66} \end{bmatrix} \tag{5.4}$$

Vetor das forças nodais equivalentes do elemento A no referencial global:

$$F_A = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \\ 0 \\ 0 \\ F_5 \\ F_6 \\ F_7 \\ F_8 \end{bmatrix} \tag{5.5}$$

Vetor das forças nodais equivalentes do elemento B no referencial global:

$$F_A = \begin{bmatrix} 0 \\ 0 \\ F_1 \\ F_2 \\ F_3 \\ F_4 \\ 0 \\ 0 \\ F_5 \\ F_6 \end{bmatrix} \tag{5.6}$$

Obtidos todos os vetores e matrizes associados a cada elemento no referencial global, basta

efetuar a sua soma para se obter tanto a matriz de rigidez como o vetor de forças nodais equivalentes global da estrutura.

A equação apresentada de seguida traduz a relação de rigidez correspondente à totalidade de graus de liberdade da estrutura

$$(K_A + K_B)\, a = (F_A + F_B) \tag{5.7}$$

Após a contabilização de todas as condições de apoio e deslocamentos conhecidos em 5.7, é possível continuar com o processo de calculo do programa. O próximo passo consiste na resolução do sistema de equações lineares resultante de 5.7 e obter todos os deslocamentos associados a cada grau de liberdade da estrutura.

Como foi já referido e explicado no Capítulo 3, depois de resolvido este sistema, é possível calcular o estado de tensão e deformação em qualquer ponto de qualquer elemento envolvido e, posteriormente, apresentar os resultados obtidos recorrendo a *software* e interfaces gráficas.

Nesta fase, assumindo que um elemento triangular dado como exemplo está presente no programa, o cálculo de estruturas mais complexas e diversificadas torna-se possível.

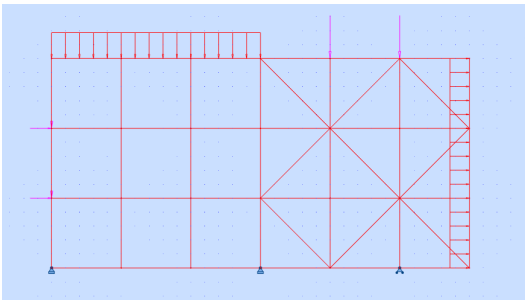

Figura 5.5: Exemplo de estrutura possível de ser calculada

Capítulo 6

Conclusões

6.1 Considerações finais

O trabalho realizado ao longo desta dissertação teve como objetivo a criação de um programa de elementos finitos e a sua consequente disponibilização numa plataforma *open source*. Este projeto foi elaborado num ambiente aberto com a finalidade de poder receber contribuições, tanto de futuros alunos da especialidade de estruturas, como de pessoas externas ao curso de engenharia civil e até da faculdade de engenharia. É importante ter em conta que o programa desenvolvido nesta fase inicial representa apenas o primeiro passo daquilo que poderá vir a ser um *software* de elementos finitos criado por estudantes da FEUP. Desta forma, para que o programa possa crescer e ter a capacidade de analisar qualquer tipo de elemento da forma mais eficiente possível, novas entradas terão de surgir no mesmo.

Numa primeira fase desta dissertação, foi abordado o assunto de *open source software*, como este funciona e como se deve lidar com projetos deste género. Quando um trabalho desta dimensão é disponibilizado para todo mundo, há certos cuidados e ações que se deve ter. Permitindo o acesso de qualquer pessoa a um determinado *source code*, surgem sempre questões legais ligadas ao uso do mesmo, assim sendo, a escolha de uma licença vai influenciar em grande parte a maneira como contribuidores interagirão com este projeto. A melhor forma de interação com quem queira participar neste trabalho é através da escrita de ficheiros de texto com o que seria bom e desejável de ser contribuído para o programa.

A estruturação deste projeto é muito provavelmente o ponto mais importante deste trabalho. Se um programa não for transversal o suficiente, a inserção de um novo elemento ou outro tipo de contribuição, não será bem sucedida. Uma das grandes chaves da diversidade de possíveis entradas no projeto foi o uso de um ficheiro *JSON* como interface de entrada e saída. A neutralidade inerente a este formato permite que surjam grandes desenvolvimentos no programa no que toca ao *input* e ao *output*. A facilidade de acesso a valores, característica deste ficheiro, possibilita também uma entrada simples de novos elementos. A estruturação de diretórios e a divisão do cálculo dos

diferentes elementos finitos até à sua assemblagem foi imprescindível para que as contribuições se possam introduzir naturalmente. A apresentação de resultados de um programa é essencialmente a fase mais relevante. Quando alguém recorre a qualquer *software* de cálculo, à partida, estará à procura de resultados, desta forma, quanto maior for a diversidade de opções de visualizações do mesmo, melhor. Assim sendo, tendo definido um armazenamento de dados de saída capaz de ser exportado para qualquer interface de análise, a visualização dos mesmos de uma forma mais tangível foi alcançada.

A admissão de contribuições num projeto requer um conjunto de regras e passos a seguir para que todo o procedimento se torne mais simples. Para tal, foram definidos todos os elementos necessários que o contribuidor deve submeter para que o novo elemento possa ser aceite e incluído no programa. Para controlar e validar as contribuições que advenham foi definida a aplicação de testes de validação de entrada de forma a que exista um filtro do que é bom código e uma contribuição funcional, daquilo que não servirá de nada e possa ser dispensado.

Com os avanços tecnológicos que se tem observado nas últimas décadas, surge cada vez mais a necessidade de adaptação a essas tecnologias de forma a tornar qualquer tarefa mais simples e automatizada. Do modo que o mundo tem evoluído, é expectável que daqui a uns anos, qualquer estudante seja obrigado a dominar uma linguagem de programação. Assim sendo, esta dissertação serviu sobretudo para que a faculdade de engenharia possa dispor de um *software* de elementos finitos em seu nome e para que qualquer aluno se sinta suficientemente motivado para aprender a programar e que, com os seus conhecimentos de estruturas e do método em causa, consiga contribuir para o mesmo programa.

6.2 Desenvolvimentos Futuros

Findada esta primeira fase do projeto, é expectável que o programa continue a ser desenvolvido de modo a que um dia, o mesmo possa ficar completo ao nível de muitos *software* já existentes. Para tal, definem-se agora alguns pontos associados a possíveis contribuições para o projeto de forma a que este possa ficar mais desenvolvido:

- Definição de novos tipos de elementos
- escrita dos programas de validacao das contribuições para o sistema de modo a que seja garantida a qualidade do sistema. Nesta fase a operação ainda não está automatizada.
- Inclusão de uma interface gráfica de entrada e saída
- Introdução de um *solver* do sistema de equações mais sofisticado
- Criação de um gerador de malha e de um pós-processador de dados
- Preparaçao do código do codigo para a introducao do calculo dinâmico

- Criação de templates para o auxílio a novas entradas. Estes templates consistem em diretórios previamente criados com toda a estruturação necessária para que, por exemplo, aquando da entrada de um novo elemento, baste apenas introduzir a informação específica do mesmo nos ficheiros em causa.

Referências

[1] Disponível em https://opensource.com/resources/what-open-source, acedido a última vez em 19 de maio de 2018.

[2] Disponível em https://opensource.org/osd-annotated, acedido a última vez em 19 de maio de 2018.

[3] Disponível em https://opensource.guide/starting-a-project/, acedido a última vez em 20 de maio de 2018.

[4] Disponível em https://www.techopedia.com/definition/8687/open-source-license, acedido a última vez em 19 de maio de 2018.

[5] Disponível em https://opensource.com/life/15/5/4-steps-creating-thriving-open-source-project, acedido a última vez em 19 de maio de 2018.

[6] Disponível em https://www.gnu.org/philosophy/free-sw.html, acedido a última vez em 19 de maio de 2018.

[7] Disponível em https://techterms.com/definition/readme, acedido a última vez em 19 de maio de 2018.

[8] Disponível em https://www.json.org/, acedido a última vez em 3 de junho de 2018.

[9] Disponível em http://softwaretestingfundamentals.com/black-box-testing/, acedido a última vez em 10 de junho de 2018.

[10] Álvaro FM Azevedo. Método dos elementos finitos. 2011.

[11] DA Dunavant. High degree efficient symmetrical gaussian quadrature rules for the triangle. *International journal for numerical methods in engineering*, 21(6):1129–1148, 1985.

[12] Carlos A Felippa. Introduction to finite element methods. *Course Notes, Department of Aerospace Engineeing Sciences, University of Colorado at Boulder, available at http://www. colorado. edu/engineering/Aerospace/CAS/courses. d/IFEM. d*, 2004.

[13] Karl Fogel. How to run a successful free software project-producing open source software. 2009.

[14] Ernest Hinton e DP Owen. Finite element programming. 1977.

[15] B Szab e I Babu. ka, finite element analysis, 1991.